Evolution Based Scheduling of Precedence Computations with Communication Costs

by

Mohammad Mohsin Nadeem

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

December, 1996

EVOLUTION BASED SCHEDULING OF PRECEDENCE COMPUTATIONS WITH COMMUNICATION COSTS

by

Mohammad Mohsin Nadeem

A Thesis Presented to the
FACULTY OF COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

Department of Computer Engineering

December 1996

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

MOHAMMAD MOHSIN NADEEM

under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee

Dr. *Mayez Al – Mouhamed (Chairman)*

Dr. *Atef Al – Najjar (Member)*

Dr. *Sadiq M. Sait (Member)*

Dr. *M.Y. Osman (Member)*

Department Chairman

Dean, College of Graduate Studies

24-12-96
Date

To those,
who are most
dear to me,
my father,
mother and
Yasmeen &
Seemi

Contents

List of Figures

List of Tables

Acknowledgements

All praises are for Allah, the Most Compassionate, the Most Merciful. May peace and blessings be upon Prophet Muhammad, and his family. I thank Almighty Allah for enabling me to complete this work.

I acknowledge the support and facilities provided by the King Fahd University of Petroleum and Minerals, and its Computer Engineering Department, for this work.

I would like to express my gratitude and appreciation for my advisor Dr. Mayez Al-Mouhamed for his guidance and support. My interest in the field of Parallel Processing is because of the courses I took with him.

I would also like to thank Dr. Sadiq M. Sait, with whom I had a long and pleasant association, for his support. My interest in Search Algorithms is the result of VLSI courses I took under him. I also thank Dr. M. Y. Osman and Dr. Atef J. Al-Najjar for their consistent support and cooperation. I was a grader for Dr. Osman and enjoyed a specially warm and sincere relationship with him. I thank him for the confidence and trust that he used to place in me.

I also thank my friends, Abbi Hasan, Abdur-Raheem, Amir Ali, Mansoor Wahid and Shahid Tanvir for their company and useful suggestions.

Finally, I thank my family, for their constant moral support and encouragement. This helped me a lot in achieving my objective.

THESIS ABSTRACT

Name: Mohammad Mohsin Nadeem

Title: EVOLUTION BASED SCHEDULING OF PRECEDENCE COMPUTATIONS WITH COMMUNICATION COSTS

Degree: MASTER OF SCIENCE

Major Field: COMPUTER ENGINEERING

Date of Degree: December 1996

Parallel Processing has become an attractive option with the decrease in the prices of single chip microprocessors and a dramatic increase in their processing power. Very fast computation requirements of a wide variety of fields have to be satisfied without the use of supercomputers as they are unavailable and costly. The single chips are also fast approaching their fundamental physical limits. The only way out is Parallel Processing. Of the parallel processing architectures Distributed Memory (DMM) Systems are an important class.

The sequential program needs to be broken down to smaller segments called tasks. These tasks need to communicate between them because once they were a part of a single sequential program. This representation of a computation is called a Directed Acyclic Graph or DAG. In it, the nodes represent the tasks and edges represent the communication between the tasks.

The assignment of tasks to processors is done by the parallelizing compiler. The schedule length or the time to finish the whole computation should be as small as possible. Finding an optimum schedule under these conditions is NP-Hard. For this reason a lot of heuristics have been developed for scheduling.

In this study we find better heuristics using the concept of Simulated Evolution. This algorithm has been successful in solving other optimization problems. This algorithm is a balance between a totally random search and a pure greedy approach. A unified framework was developed which can be used to study the scheduling problem in a better manner. The framework allows us to display the task graph, schedule the task graph using the heuristic specified by the user and then display the resulting schedule in the form of a Gantt Chart. This environment is comprehensive and can be used in the investigation of the nature of different heuristics.

Keywords: Schedule, Processor, Task Graph, Simulated Evolution & NP-Hard.

King Fahd University of Petroleum and Minerals, Dhahran.

December 1996

موجز الرسالة

اسم الطالـــب : محمد محسن نديم

عنوان الرسالة : جدولة افضليات الحسابات المعتمدة على التطور مع وجود كلفة اتصالات .

التخصـــص : هندسة الحاسب الآلي

تاريخ الشهادة : ديسمبر ١٩٩٦م .

لقد أصبحت المعالجة المتوازية تخصصًا جذابًا مع تناقص اسعار شرائح الميكروبروسيسور والتزايد الحائل في طاقة المعالجة لهذه الشرائح . فلقد أصبحت الحاجة ملحة للحسابات العالية السرعة في كثير من التخصصات من دون اللجوء الى استخدام الحاسبات الفائقة نظراً لكلفتها العالية ولعدم توفرها .

ولقد أوشكت الشرائح الأمادية أيضًا على الوصول الى حدها من حيث السرعة . وأصبح المخرج الوحيد لهذه المشكلة هو المعالجة المتوازية ، حيث تعتبر أنظمة المعالجة الموزعة هي صنف هام من أصناف بناءات المعالجة المتوازية .

يتم تعيين الوظائف للمعالجات في المعالجة المتوازية بواسطة مجمّع متوازي ، حيث يجنب أن يكون الوقت اللازم لانهاء الحساب أصغر ما يمكن ، ولقد تم تطوير كثير من الاساليب التجريبية الحدسيه لجدولة الوظائف في المعالجة المتوازية ، وفي هذه الدراسة ، فإننا نتوصل الى اساليب تجريبية أفضل باستخدام مبدأ محاكاة التطور .

ولقد نجح هذه الخوارزمي في حل مشاكل أخرى للوصول الى الحل الأمثلي ، ويُعتبر هذه الخوارزمي عبارة عن توازن بين البحث العشوائي الكامل والاسلوب الذي لا يقبل الخسارة .

ولقد تم تطوير إطار موحد يمكن استخدامه لدراسة مشكلة الجدولة بطريقة أفضل . حيث يسمح هذا الاطار بعرض الرسم البياني للوظيفة ، وجدولة الرسم البياني للوظيفة باستخدام الاسلوب التجريبي المحدّد من قِبَل المستخدم ، وعرض الجدول الناتج بشكل خط بيان جانت .

ويتضمن هذا الخوارزمي بأنه سهل الفهم ويمكن استخدامه في التحقق من طبيعة الأساليب التجريبية المتعددة . اصطلاحات : جدولة ، معالج ، خط بيان الوظيفة ، محاكاة التطور . واسلوب معقد من نوع NP .

درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران ، المملكة العربية السعودية
ديسمبر ١٩٩٦م

Chapter 1

Introduction

Parallel Processing has become an attractive option for very large and complex computations. Previously only vector supercomputers were being used for this purpose. These super computers are costly and are not available easily as the nations that build them guard these computers as military secrets. The reason for this is that these computers can easily be used for military projects. Now parallel processing provides a way by which we can speedup long and complex computations for a variety of different purposes.

The reason for the sudden spurt in the popularity of parallel processing are twofold. Firstly, there are now available cheap, fast, and reliable microprocessors and their price has been going down continuously and computational power increasing. Secondly, the software sizes have grown in such a way that traditional single processor computation has become unattractive. The software sizes for problems

such as numerical weather forecasting, fluid flow problems, handling of geophysical data, genetic engineering, real-time analysis of military and medical data, fast fourier transforms, graphics based applications, artificial intelligence and automation, logic simulation etc. have increased tremendously.

Most of the above problems have programs that are compiled once and then are executed many times over different data sets. Keeping this in mind it is imperative for a parallel computer to have a very efficient operating system. The operating system's parallelizing compiler is responsible for distributing the workload of the problem to different processors of the multiprocessor. This distribution should be such that the semantic flow of the problem remains undisturbed. The problem for this purpose is thought to be composed of constituent tasks. Each task can not be sub-divided further and is to be executed on a single machine only. Each task after finishing may provide other tasks of the problem some data. Also each task can only begin if it has the data needed for its computation.

The above model of a parallel problem is called a directed acyclic graph, *(DAG)* in which nodes represent tasks and edges represent communication arcs. So a *DAG* is composed of a set of tasks with message communicating edges between them. Each message edge can have a particular amount of data to be conveyed. Similarly, each task can have a particular amount of computation required. The communication edges enforce a precedence constraint on the task as a task needing data from some other task can not begin until and unless it has obtained all its needed data from

earlier tasks.

Now one of the jobs of the compiler for parallel machine is how to distribute the tasks amongst different processors; this is called a schedule. This distribution should be such that the overall finish time of the problem is minimized. Some effort should be applied to find a near optimum schedule as we will run the compiled program many times over different data sets. Unfortunately, finding an optimum schedule is computationally expensive since the scheduling problem is NP-complete [1]. For finding an acceptable schedule many methods have been used. Some are heuristic based while others are search based. There are different advantages for both of them. The search based methods are simulated annealing, genetic algorithm, tabu search and branch & bound methods. They promise a better quality of solution though computationally they are very expensive. We can defend their use because they can repay the initial effort when a slightly better solution is run thousands of times. The heuristic methods are computationally cheap though they may not provide as good a solution. Some such techniques are earliest task first (ETF), dominant sequence clustering (DSC) and task duplication (TD). Further discussion about these algorithms is presented in Chapter 4.

The heuristic techniques try to minimize a certain quantity at each scheduling step in the hope of minimizing the overall schedule. For example, ETF tries to minimize the idle time on each processor. This in a way is a local approach. A slightly different approach is to use a priority based scheduling. Here we have an

estimate of each task's importance and this is also taken into account at each decision step. By doing this we incorporate a global view also at each step of the technique.

In this thesis we have developed a new search based priority scheduling. The search algorithm used is Simulated Evolution. This is a new search technique and is very promising. Its advantage lies in the fact that it does not create totally random solutions and then select from among them. Instead it does a directed search and searches only among better solutions. Each solution is generated by a priority based scheduling. How we move in the search space is controlled by simulated evolution. This is in contrast to genetic algorithm and simulated annealing where the solutions are generated totally randomly. This just increases the run-time of the algorithms. By using simulated evolution we find a good schedule in an acceptable number of iterations. We will discuss simulated evolution in Chapter 3.

Another method called the Arc Addition Method was devised. This is explained in Chapter 5. We also give the results of empirical testing of this method using a sample of 1470 randomly generated task graphs. The task graphs have varying degrees of granularity and parallelism.

The new method, called the Mobility Based Evolution Scheduling is presented in Chapter 6. This technique again, is tested by using the same set of graphs as for the Arc Addition Method. The results are presented in the form of graphs and tables.

In Chapter 7 we give the conclusions about this MS thesis and discuss future

work in this area.

1.1 Parallel Processing

The computation power which can be derived form a single chip is limited. These limitations are fundamentally physical. The chip has a finite size and it takes a finite amount of time for an electrical signal to cross an electrical path. Moreover these paths are in silicon and so the speed of signal transmissions is still more slowed down. For example if the diameter of a chip is $3cm$ and speed of electrical signal propogation in silicon is $3 \times 10^7 m/s$ then for end to end signal propagation we need $3 \times 10^{-2}/3 \times 10^7 = 1 \times 10^{-9} seconds$. If we can do one floating point operation (FLOP) in this time then we can reach up to 1 GFLOPS. Currently DEC's Alpha processor operates at 0.1 GFLOPS. Moreover, the chip circuitry is becoming more and more complex and the path lengths in the chips are increasing. So even though the dimensions of semiconductor devices are decreasing there is a limit to what we can get from a single chip.

We know that most of the scientific problems have an inherent parallelism. By this we mean that not all parts of a problem need to be computed one after the other. Many of them can be done in parallel on different processing units. This reduces the finish time of our problem. Thus parallel processing tries to take advantage of the inherent parallelism of a problem to solve it in a quicker way by using much more

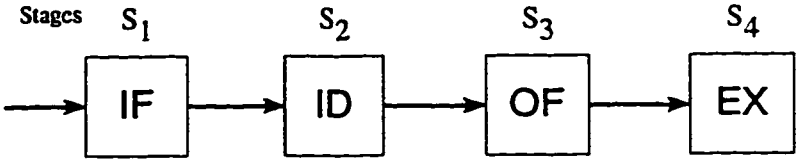


Figure 1.1: A pipelined processor.

hardware.

1.2 Parallel Processing Architectures

There are basically three types of parallel processing architectures [2]. The first is called the Pipeline Architecture. The pipeline computer tries to exploit the *temporal parallelism* of a problem. This is done basically at the instruction level. There are four steps for each instruction. They are *instruction fetch* (IF) where the instruction is brought from the main memory; *instruction decode* (ID) which identifies what operation each instruction has to do: *operand fetch* (OF) which brings the data needed for execution of the instruction: and lastly the *execution* (EX) where the actual operation required by the instruction is done. These stages are arranged in the form of a cascade and successive instructions are executed in an overlapped manner. A pipelined computer can be represented as shown in Figure 1.1. These machines are better at performing the same operation repeatedly on different components (data). They are therefore more attractive for vector processing.

The pipelined processor executes consecutive instructions in an overlapped man-

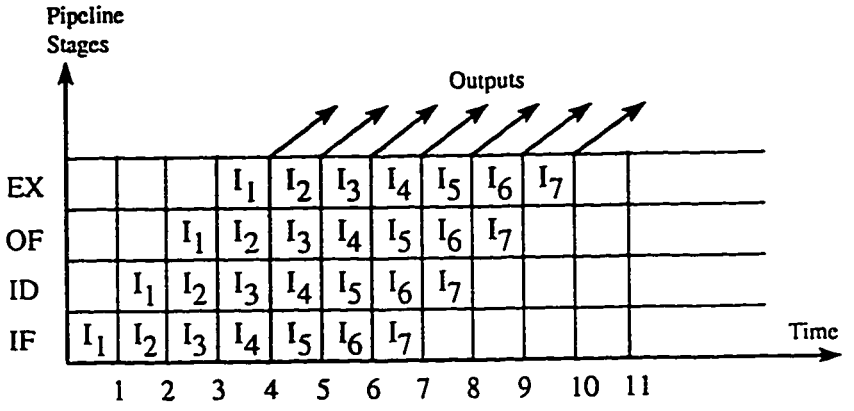


Figure 1.2: Instruction execution on a pipelined processor.

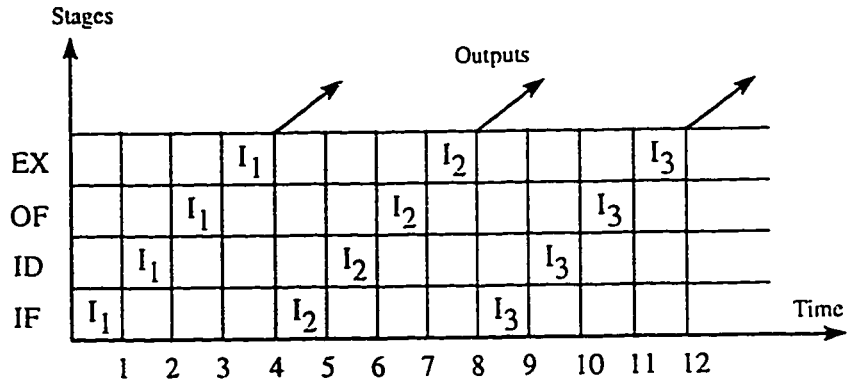


Figure 1.3: Instruction execution on a non-pipelined processor.

ner and hence a higher speed, than available with a non-pipelined processor, is achieved. This is shown in Figure 1.2. The same instructions are then executed by a non-pipelined processor. This is shown in Figure 1.3. We can easily see from these figures that a pipelined processor is much faster.

The second architecture is a called an *array processor architecture.* This architecture tries to exploit the *spatial parallelism* of the problem. This consists of an array of processing elements performing the same operation over different data sets. Each PE consists of an ALU with registers and a local memory. The PEs are connected by a data routing network. They are used for image processing applications and vector algebra [2]. These problems have an inherent regularity. The computations proceed in a predetermined manner and result in high performance using pipelining and exploiting parallelism. They are further classified as *wavefront arrays* and *systolic arrays.* In systolic arrays the control is synchronous while in wavefront arrays it is asynchronous.

The processor arrays are used to map algorithms for general problems into hardware. The problem is represented in the form of a data flow graph. One such graph is shown in Figure 1.4. This dataflow graph from [3], represents the following expressions.

$$A = F\frac{k - MW^2}{(k - MW^2)^2 + W^2C^2}$$

$$B = F\frac{WC}{(k - MW^2)^2 + W^2C^2}$$

The resulting data flow graph is then mapped onto a regularly connected array of

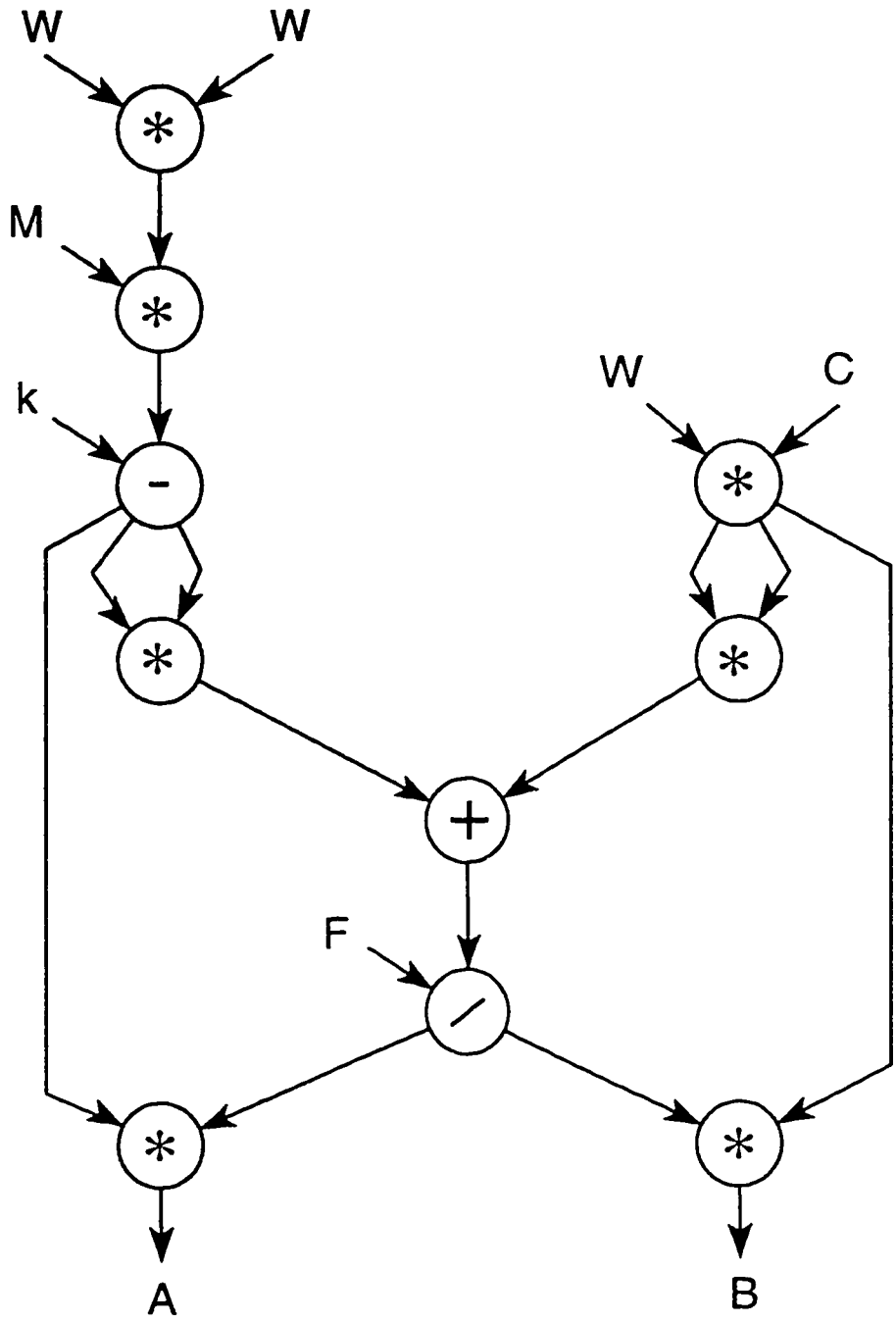


Figure 1.4: An example of a data flow graph

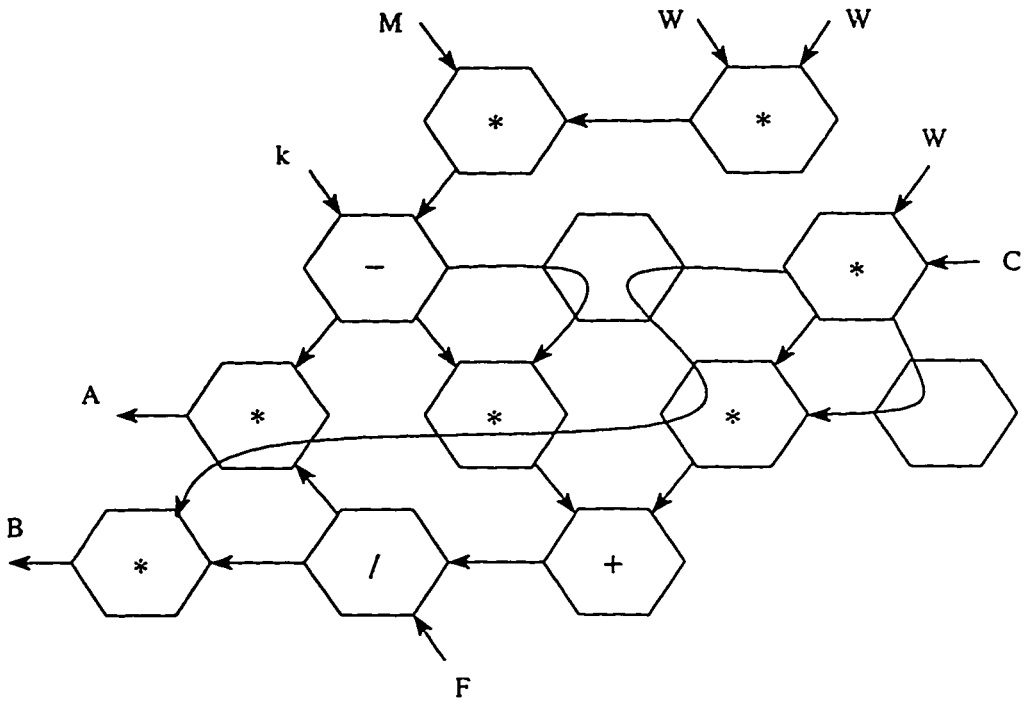


Figure 1.5: Mapping of the dataflow graph onto a hexagonally connected array of processing elements.

processors. The processors here are much simpler and are called *Processing Elements.* Each *PE* has a capacity for communication and doing simple arithmetic operations. The memory consists of a few registers. The mapping of the dataflow graph shown in Figure 1.4 to a hexagonally connected processor array is shown in Figure 1.5.

The third type is called a *multiprocessor.* In this there are more than one full processor which can be connected through switches, buses or multiport memories. They can have a shared memory or each one can have a separate memory. The processors here have much more functionality compared to that of the processing elements of processor arrays. These are used for problems which are not as regular as the problems for which processor arrays are used. They are thus used for problems which can be expressed as any Directed Acyclic Graph and it need not have a pattern or regularity in it. Most multiprocessors fall in a class called the Multiple Instruction Multiple Data type or MIMD [2]. In this multiprocessor each processor has its own private memory and all processors are linked by a communications network. The topology of the network can be a regular one like ring, hypercube or fully-connected or it can be arbitrary. This is also known by the name Distributed Memory Multiprocessor or DMM. Figure 1.6 shows the functional design of a distributed memory multiprocessor [2]. The tasks to each processor are assigned by the scheduler of the parallel compiler. Another term describing such systems is the loosely coupled MIMD. Here loose coupling means that the interaction between

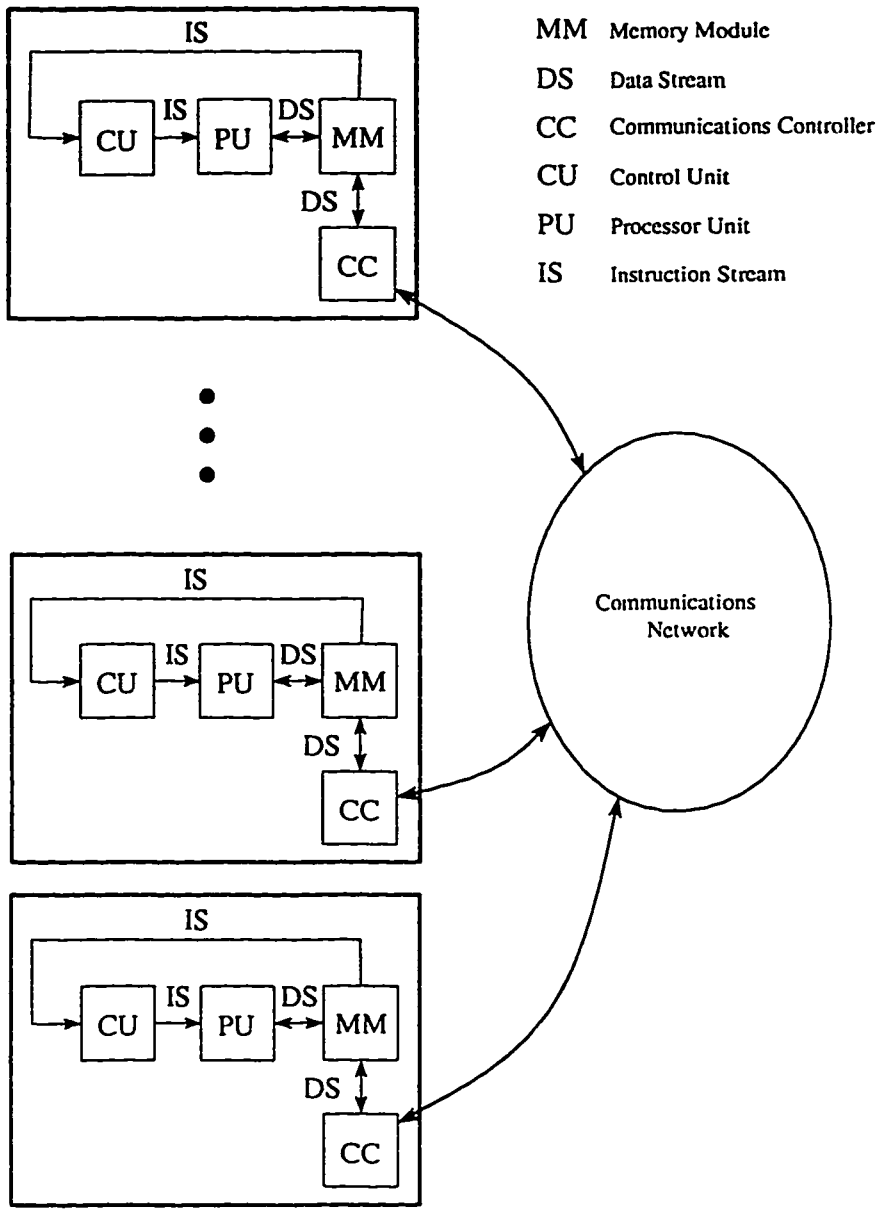


Figure 1.6: Functional structure of a distributed memory multiprocessor.

different processors is not much.

In tightly coupled MIMD systems the interaction between processors is more and they tend to be classified under processor arrays. The tightly coupled multiprocessor fall under the class of Shared Memory systems. In these multiprocessors, though there are separate processors, the memory is shared between different processors. Because of this reason they can interact to a higher degree as the cost of interaction is low. This is also the reason why such multiprocessors are known as tightly coupled systems. A shared memory multiprocessor is shown in Figure 1.7.

The speedup achieved by a parallel computer should ideally be equal to n if n processors are present in the multiprocessor. But this is generally not the case. In every problem there are some parts which have to be serialized. Because of this portion the potential speedup is limited. This fraction is called the *Amdahl's Fraction* and the law is known as *Amdahl's Law*. For example if 5% of a problem is non-parallelizable and 95% is fully parallelizable then the upper limit to the speedup is $1/0.05 = 20$. So even if we employ more than 20 processors we will not be able to get a better speedup. Moreover there are delays in the communication network and conflicts in memory accesses. So the actual speedup is lower than n even though we have n identical processors.

In this chapter we presented an introduction to parallel processing. We talked about its need and which types of computer architectures are used to do parallel processing. We talked about Pipelined Processors, Array Processors and Multipro-

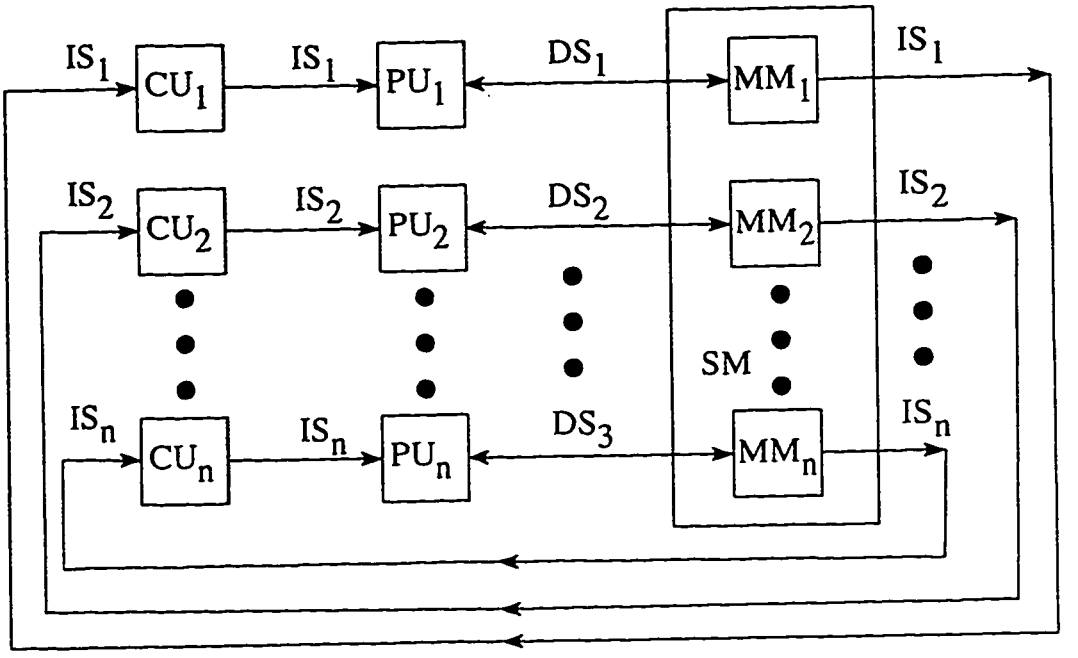


Figure 1.7: Functional structure of a shared memory multiprocessor.

cessors. We also discussed in brief the Amdahl's Law regarding the upper limit to potential speedup.

Chapter 2

Problem Definition

In this chapter we define the models that are used in literature to properly represent the scheduling problem. We will describe the notation for the DAG, for the multiprocessor and other quantitities used in the discussion. The problem and the objectives that are addressed in this thesis are then introduced.

2.1 The Graph

There are two slightly different models for the DAG. The first model was proposed by Hwang [4] and considers that some finite communication time is required in sending some units of data from one processor to another through the communication network. This model thus takes into consideration the inter-task communication costs. The second model, by Graham [5] is a simplification of the previous model.

Here all the inter-task communication times are assumed to be zero. Though, this is mostly unrealistic and can be true only in some particular environments where the communication time is very much smaller in comparison to computation time. We use the first model and consider the inter-task communication times.

The parallel computation is represented as a directed acyclic graph or DAG. The entire problem is thought to be composed of a set Γ of m tasks $(T_1, T_2, \ldots, T_{m-1}, T_m)$. The tasks have precedence relationships amongst them. The precedence arcs represent inter-task communication. More over, each task needs a specific amount of computation time from a processor, and this is called the task's computation time.

Now each node in the graph represents a task, $T \in \Gamma$, each edge represents a precedence constraint between a parent (predecessor) task and its child (successor). Edges $T \rightarrow T_i$ are weighted and the weight represents the units of data needed to be passed from the parent to the child. The number on edge is $c(T, T_i)$ and denotes that task T sends these many units of data to task T_i upon its completion. The whole set of inter-task communications is denoted by C. The edges have a definite direction (downwards) and no cycles are present. In each node of the graph we have the name of the task i.e.. T and its computation time, represented as $\mu(T)$. We can also associate an exit node and an entry node with each graph without any loss of generality. An entry node is a dummy node having no predecessors and having $\mu(T) = 0$, same is the case with the exit node. The edges connecting these nodes with other nodes are of weight zero. We can reach any node of the graph from

the entry node and from each node of the graph we can reach the exit node. Thus any precedence constrained computation with communication costs being taken into account can be represented by the quadruple $G(\Gamma, \rightarrow, \mu, C)$ and it is called the task model. An example graph is shown in Figure 2.1.

2.2 The Multiprocessor

Now we describe the multiprocessor model. We have already said that our multiprocessor is a Distributed Memory Multiprocessor and each processor has a separate memory. The processors as well as the memories are identical. The messages $c(T, T_i)$ are passed via the communication network. Even though there are logical links between all the processors in the multiprocessor, the lengths of the links can be different. This depends on the topology of the multiprocessor. This system is denoted by $S(P, R)$. P represents the set of n identical processors and R represents the inter-processor routing cost. The time to transmit one unit of data from processor p to processor p' is given by $r(p, p')$. As we have already said, the interconnection network denoted by R can be arbitrary or regular. The interconnection network is represented by an $m \times m$ matrix if there are m processors in the multiprocessor. The entries of the matrix are $r(p, p')$. If a processor p is not connected to processor p' then the routing cost entry is ∞. If the routing costs are high it can model a loosely coupled system, on the other hand, if they are very low, it can model a

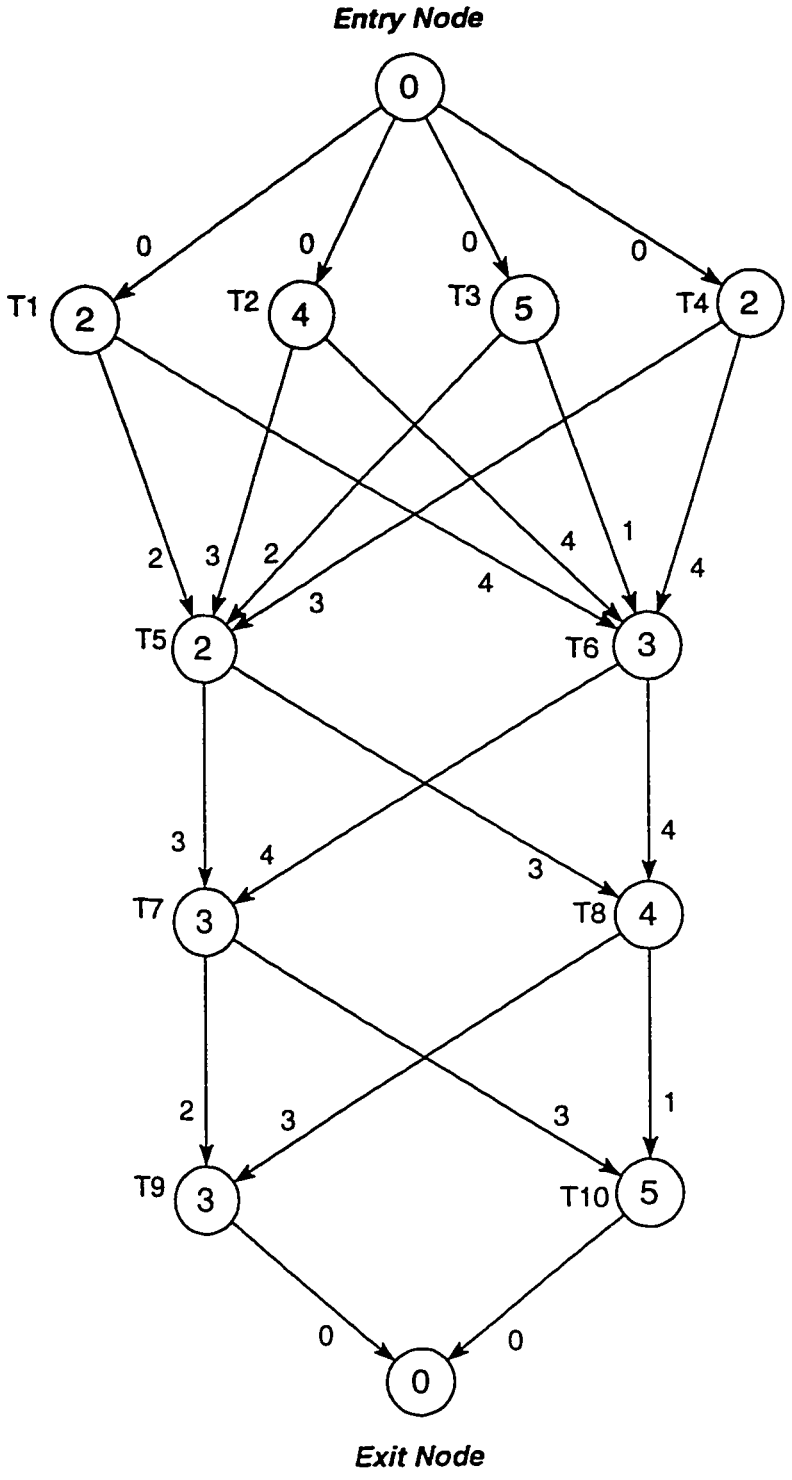
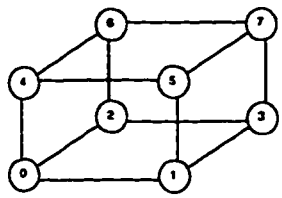


Figure 2.1: A task graph with communication.



	0	1	2	3	4	5	6	7
0	0	1	1	2	1	2	2	3
1	1	0	2	1	2	1	3	2
2	1	2	0	1	2	3	1	2
3	2	1	1	0	3	2	2	1
4	1	2	2	3	0	1	1	2
5	2	1	3	2	1	0	2	1
6	2	3	1	2	1	2	0	1
7	3	2	2	1	2	1	1	0

Figure 2.2: A hypercube multiprocessor with 8 nodes and its interconnection matrix.

tightly coupled system. Interconnection patterns such as ring, fully-connected, and hypercube can be easily represented. A hypercube multiprocessor with 8 nodes is shown along with the interconnection matrix in Figure 2.2

To recapitulate, let us describe in brief the over all graph and multiprocessor models. T denotes a task and with it is associated a computation time given by $\mu(T)$. The processor on which task T is scheduled is given by $p(T)$. The amount of data sent from T to T' is given by $c(T,T')$. The communication cost or the distance between processor p_1 and p_2 is given by $r(p_1,p_2)$.

2.3 Scheduling Terminology

Here we will explain and define some common terms which will be used frequently in later chapters.

- **The Path Length**

  There are two definitions of path length. One definition assumes that we take

into account the communication costs i.e., the model $(\Gamma, \rightarrow, \mu, C)$ while the other assumes the communication costs to be zero or the model $(\Gamma, \rightarrow, \mu)$. If the communication costs are disregarded then the path length becomes a static quantity. This means that it can be obtained from the graph itself and it will not depend on the schedule. The path length between two tasks T and T' in this case is defined as the sum of computations along the path between them. If there are more than one path then we have more than one path lengths and we can take the larger value as the overall path length between T and T'. If the computations are measured in seconds then the path length is also in seconds.

For the model $(\Gamma, \rightarrow, \mu, C)$ the path length is a dynamic quantity and depends on the way the task graph is mapped onto the multiprocessor. So it depends on the graph, the schedule and the multiprocessor. The path length here measures how much time it takes to reach from start of task T to end of task T' in a given schedule on a given multiprocessor. If the two tasks lie on the same processor and T is the immediate predecessor of T' then no time will be lost in conveying the data $c(T, T')$ and the path length will be $\mu(T) + \mu(T')$ time units. Else if they are on different processors i.e. $p(T) \neq p(T')$ then path length will be $\mu(T) + c(T, T') \times r(p(T), p(T')) + \mu(T')$ time units. So we see that if the mapping of tasks to the processor changes or the interconnection network changes then we will have a changed path length.

There is another static approach where we add the edge weights $c(T, T')$ to the tasks computations. This is again independent of the schedule and the multiprocessor.

- **The Critical Path**

  The critical path is the measure of the length of a complete graph G and is denoted by CP. Thus it is the longest path length for a graph from the entry node down to exit node. Like the path length we have two definitions for CP. For the model, $(\Gamma, \rightarrow, \mu)$, where we disregard communication costs the CP is a static quantity and can be found from the graph itself. On the other hand for the model $(\Gamma, \rightarrow, \mu, C)$, the CP is a dynamic quantity and depends on a given schedule and on a given multiprocessor. The explanation is the same as for the path length described earlier.

- **The Task Level**

  The task level of a task T is referred by $l(T)$. This measures the longest distance of the task from its start to the exit node. This in a way tells how much computation and/or communication is left after this task and consequently its criticality to the over all computation. Again we have two definitions for the two different models. For the model $(\Gamma, \rightarrow, \mu)$, task level is a fixed quantity and depends only on the graph. Whereas for the model $(\Gamma, \rightarrow, \mu, C)$, task level is a dynamic quantity and keeps changing depending on the multiprocessor and

the given schedule.

- **The Ready to Run Set**

  While building a schedule the tasks which have all their predecessors already assigned are called the ready to run tasks. The set of all tasks which are ready to run is denoted by RTR. Initially the RTR contains all those tasks which are immediate successors of the entry task. At each scheduling step one task from the RTR set is scheduled and then taken out of RTR. We also update the RTR if any new tasks become available with all their predecessors assigned.

- **The Earliest Starting Time**

  Any tasks T enters the RTR when all its predecessors denoted by $T_i \in pred(T)$ have already been assigned. Now this tasks can begin only after all of them have finished execution. Some additional time delay may also be incurred to transfer the data needed by T from one of its predecessors. The earliest starting time of T is denoted by $est(T)$. If we also wish to refer to the processor on which this time was found, we use the notation $est(T.p)$.

- **The Task Completion Time**

  The completion time of a task is simply the sum of its earliest starting time and its computation time. This is denoted by $ct(T) = est(T) + \mu(T)$.

2.4 The Scheduling Problem

The scheduling problem is encountered in many different fields. We are well acquainted with one such field. This is the scheduling of computation graphs on multiprocessors. The scheduling problem is also encountered in job shop scheduling in industrial engineering. This is a slight variation from our task graph problem. In its most general form its complexity is NP-complete [1], [6]. Even in simplified versions of the problem like scheduling one or two time unit tasks on two processors is also NP-complete. But there are two cases where we can find optimal time schedules with polynomial time complexity if we disregard inter-task communication times. These cases are:

- scheduling tree-structured graphs with equal execution times on any multiprocessor system [7] and
- scheduling arbitrary graphs with equal execution times on a two processor system [8], [9].

The optimal algorithms for both these cases use the static task level to estimate the priority of each task in the task graph. If we incorporate inter-task communication in any one of the above cases then the problem becomes NP-complete.

2.5 Scheduling Approach Classification

We can classify different scheduling approaches on the following basis.

- **Local vs Global**

  Local scheduling deals with how to allocate time slots on a single processor to different processes of a multi-threaded environment. This is required when single processor machines are used in multi-user modes like UNIX, Windows'95, Novell etc. Here we are only dealing with global scheduling where we have to assign tasks to processors of a multiprocessor.

- **Static vs Dynamic**

  In static scheduling all the information about the problem i.e., the task graph is fixed beforehand. The multiprocessor is also fixed and from the schedule we can find the start times and processors for each task. This approach fails if the task graph can change depending on some conditions triggered by a specific instance of data. Conditional branches and loops can cause this non-determinism. In dynamic scheduling we do not know which tasks will need to be executed beforehand. If a certain condition is true then we may need to execute some task. So when a task needs to be executed then a decision will be made on the fly about its scheduling and its processor assignment. This decision making incurs some time delay and moreover now we can not try to find optimum or near optimum schedules because all our decisions are based

on the conditions at a particular time instance and therefore we lack the global view of the problem.

- **Adaptive vs Non-adaptive**

  This classification criterion is only with respect to dynamic scheduling. In Adaptive scheduling the scheduler tries to use the feedback gained from previous runs of the program to generate a better schedule next time. So an adaptive scheduler may collect data about the run time needed by the program and then try to change its behavior depending on the feedback information. Non-adaptive scheduler on the other hand is not so intelligent and it will do the same thing irrespective of the number of times the same program has been scheduled by it before.

- **Non-preemptive vs Preemptive**

  In non-preemptive scheduling once a task has started its execution it is not stopped until it has finished execution. So at a time each processor is dealing with only one task. On the other hand in preemptive scheduling a task which the scheduler considers to be more important can suspend a less important task temporarily. The suspended task can again start later from the point where it was suspended.

- **Contention based system vs Contention free system**

  The communication network can create conflicts if the same links need to be

used by more than one processor at the same time. This contention can produce delays which may not be estimated beforehand. In a contention free system there can be no conflicts in transmission of inter-task data and there is enough bandwidth available for this. So only a fixed amount of time is needed for passing one unit of data from one processor to another. This is fixed and is independent of the network conditions and thus is known beforehand. In contention free systems we also assume that communication and computation can proceed in a time overlapped manner. This in essence means that communication is controlled by another device besides the main processor.

In this thesis we assume a static, non-preemptive scheduling for a contention free multiprocessor system.

2.6 Thesis Objective

We will deal with evolution based scheduling of general precedence constrained task graphs with communications on distributed memory multiprocessors. The graph problem will be represented by a DAG or Directed Acyclic Graph. The multiprocessor will be modeled by $S(P, R)$ and different topologies of interconnection will be considered like hypercube, ring, fully-connected etc. The communications network will be assumed to be contention free and the scheduling will be static and non-preemptive.

Under the above conditions the objective will be to minimize the overall finish time of computation graphs on the multiprocessor. We will use ETF scheduling modified to take into account the priority of each task. This means that our decision regarding which task to start will be governed by the priority or the task level of each task as well as its earliest starting time. The two will be combined to form a composite decision function called $d(T)$ for task T. The priorities will be found using a new deterministic concept and they will be handled and updated using the simulated evolution concept. The mixing of the two should give good results because it includes the intelligence of heuristic methods and the hill climbing feature of search based methods. The twin objectives are :

- Finding a better criterion for the priority of each task to utilize it in the development of the composite decision function and
- Using the general form of the simulated evolution approach to search in the search space of good solutions to generate acceptable quality schedules.

Chapter 3

Simulated Evolution

It is well known that NP-hard complexity problems occur very frequently in many fields of science and engineering. Many of such problems are the traditional travelling salesman type, bipartite matching, placement of cells on a chip surface, network bisection, scheduling, high level synthesis etc. For all these problems we have to content ourselves with less than an optimal solution. The search space is very large and exhaustive searches can only be done for very small problems. For this reason heuristics are used to prune the search space so that we search in a directed manner and move towards the global optimum in our walk through it. Because of the tremendous computational effort required in reaching the global optimum we stop if the solution satisfies our constraints. Also it is known that greedy heuristics easily get stuck in a local minima because of their very nature. The advantage with constructive or constructive-iterative techniques is that they can produce an

acceptable solution quickly. But if we wish to improve the solution quality we get nowhere with them. They always try a greedy approach with respect to their cost functions and end up with less than an optimal solution. Because of this reason, hill climbing heuristics like *Simulated Annealing*, *Genetic Algorithm* and *Tabu Search* etc. produce much better solutions though they may take more computational time to generate such solutions.

3.1 The Algorithm

This is a recently proposed algorithm for combinatorial optimization problems. This algorithm mimics the natural evolution of biological species. We know that species continue to evolve under various constraints and become fitter and fitter with respect to their environment. During this process of *Natural Selection* only fitter individuals of a population are allowed to pass on their favorable characteristics to the next generation. The individuals with unfavorable characteristics die without passing their disadvantageous characteristics to future generations. This process now results in better and better populations as time progresses. Some times a trait which was not present in any of the individuals of the parent population suddenly appears in the offspring. This process of introduction of new traits in a population is called *mutation*. Some times mutations lead to a completely new species much better adapted to the environment then its predecessor species. The predecessor species

may die out altogether.

We now describe Simulated Annealing and Genetic Algorithm so that we can compare and contrast their features with Simulated Evolution.

3.2 The Genetic Algorithm

Genetic Algorithm was developed by Holland [10]. In Genetic Algorithm there are a number of solutions and each represents one solution of the problem. The algorithm can start with random solutions and the set of these solutions are called the *population.* Each solution is represented in the form of a string of symbols, called *genes.* The string made up of genes is called a *chromosome.*

The solutions in a population interact among each other via chromosomes through crossover operators. New characteristics are introduced via mutation operator. The solutions on which these operators are applied are called parents and are chosen from amongst the population probabilistically depending on their fitness. This results in the development of a new set of solutions called the *offsprings.* Now we again select from the combined population a population of the initial size. This selection is again probabilistic and depends on fitness. Measurement of fitness is usually based on the objective function. So in the scheduling context it can be the finish time of a schedule. Those individuals (solutions) of the population which have low fitness value are given lesser chance of moving to the next generation. The important point to note is

that the selection for next generation is not deterministic, rather it is probabilistic. This means that solutions with low fitness can also be selected but their selection probability is small and is proportional to their low fitness. Another point to note is that new solutions are generated by only crossover and mutation operators. Both of them operate in a totally random manner and we do not invest any effort in building up a solution using any intelligence of our own. The disadvantage is that this increases the search space even to those regions which are unlikely to yield a good solution. The general framework of Genetic Algorithm is given below [11].

```
Genetic Algorithm:
    Begin
        Make_Population:
        /* Generate initial population randomly of size N_i */
        For i := 1 to N_i do
            Calculate_Fitness(Population[i]);
        For i := 1 to N_g do /* N_g is total number of generations */
        Begin
            For j := 1 to N_o /* N_o is total number of offsprings */
            Begin
            (A,B) := Select_Parents;
            Offspring[j] := Do_crossover(A,B);
            Do_Mutation(Offspring[j]);
            Calculate_Fitness(Offspring[j]);
            End;
        Save_Best;
        Population := Select(Population,Offspring,N_p);
        End;
        Return_Best;
    End.
```

The number of initial solutions are referred by the term N_i. We apply crossover

operator on two parents (solutions) referred to by A and B. These parents are again selected probabilistically depending on their fitness. This means that though there are more chances of selecting better solutions we will also select some not so good ones also because they may later lead to optimum solution. The number N_i governs how parallel our search is. After doing crossover we generate another new offspring. We now apply the mutation operator. This is done with a low probability. If it is very high there will be lot of randomness and it will be more like a random search. The resulting solution is then evaluated for its fitness. This way we generate N_o offsprings. We always store the best solution. This process is repeated N_g times. The steps are:

1. Initialization: An initial population of random solutions is generated.

2. Evaluation: The objective function for each member of population is evaluated.

3. Genetic Operations: New search solutions are generated by applying genetic operators to the current population.

4. Steps 2 and 3 are repeated until the algorithm converges.

The Genetic algorithm has been used for many NP-hard problems successfully. It has also been used for multiprocessor scheduling in [12] and [13].

3.3 The Simulated Annealing

Simulated Annealing was first proposed by Kirk-patrick, Gelatt and Vecchi in 1983 [14]. This algorithm imitates the process of annealing metals. Annealing is done to impart a good crystal structure to metals. This is done by first heating the metal piece to a high temperature and then allowing it to cool at a precisely controlled rate. When the metal is heated the atoms gain enough energy and break their bonds and hence destroy the crystal structure.

When cooling starts the agitated atoms get slowed down. Now the rate of cooling is such that the atoms fall into their correct locations to attain a good crystal structure. This is not possible if the rate of cooling is abrupt. The temperature is decreased by a step and the metal sample is kept at this temperature for a certain amount of time. This gives opportunity for atoms to get into a minimum energy position for this temperature. At higher temperatures larger movement is allowed to the atoms but as the temperature decreases the movement also becomes small and small. At zero temperature we expect the atoms to have settled in their perfect crystal positions and because of no more energy no movements are allowed and the crystal shape remains undisturbed. A good crystal structure is analogous to a good solution in the case of combinatorial optimization. This is because in a good crystal structure each atom is in a minimum energy position with regards to its neighbouring atoms. When all the atoms of a crystal are placed in their proper

positions, the overall energy of the crystal system decreases and we find an optimum structure.

The movement through the search space is done via the *neighbor function.* The neighbor function operates by creating a random change in the current solution. This is called neighbor function because we cannot jump from one solution to an entirely different solution. Rather we make only a slight change from the present solution. This slight change though may result in a large change in the objective function of the solution. The objective function for a solution is referred to as its cost. We now give the algorithm [11], and then explain it.

Simulated Annealing;
Begin
$T := T_0$; /* Temperature T is initialized to T_0 */
$S := S_0$; /* Solution S is initialized to S_0 */
$Time := 0$;
Repeat
Metropolis(S, T, M); /* M represents time to spend at temperature T in Metropolis */
Save_Best:
$Time := Time + M$: /* update total time spent */
$T := \alpha \times T$: /* decrease temperature by factor α */
$M := \beta \times M$: /* increase time in Metropolis by factor β */
Until$(Time \geq MaxTime)$; /* Time allowed is $MaxTime$ */
Return_Best:
End.

Procedure Metropolis(S, T, M);
Begin
Repeat
$NewSolution$:= **Neighbor**(S);
Δh := **Cost**($NewSolution$) - **Cost**(S);
If (($\Delta h \leq 0$) **or** (random$(0,1) \leq e^{-\Delta h/T}$)) **then**
$S := NewSolution$;
$M := M - 1$;
Until ($M = 0$);
End.

Firstly, we initialize the solution to S_0. This initial solution may be randomly generated or it may be generated by using a computationally cheap deterministic algorithm. Next we set the initial temperature as T_0. The setting of this parameter is done by trial and error. The time spent in the algorithm is initialized to zero. The heart of the annealing algorithm is the Metropolis procedure.

This procedure takes as input a starting solution S, a given temperature T, and a time duration M. The main program just does the job of updating parameters Time and Temperature. The factor α is a real number less than 1 which does the job of reducing the temperature. The temperature is reduced in smaller and smaller steps in the form of a geometric progression. The factor β does the job of controlling how much time we spend at each given temperature. As the temperature is lowered we spend greater and greater time at each step. Thus β is a number greater than 1.

The variable $Time$ keeps control over how much time we are spending in the annealing algorithm. The maximum allowed time is $MaxTime$. In the procedure

metropolis we generate new solutions via the neighbor function.

After each generation the cost of this new solution is found. If the cost is lesser, we unconditionally accept the new solution. If the cost is higher, the acceptance is probabilistic. This is governed by the function $e^{-\Delta h/T}$. At high values of temperature because of this function we accept even bad moves. As the temperature reduces the selection becomes greedier and when temperature is 0 we do not accept bad moves at all. The random number generator is assumed to generate random numbers between 0 and 1 with uniform probability distribution.

The main problem with this algorithm is that the temperature parameter is difficult to control. Also the run-time of the algorithm is high. Again we see that when making moves via the neighbor function we do not use any intelligence or heuristic. This is done totally randomly. The Simulated Evolution algorithm allows us to use some intelligence while generating new solutions.

3.4 The Simulated Evolution

This algorithm was proposed by Kling and Banerjee [15], [16]. This was applied to the problem of standard cell placement in VLSI. This heuristic is again based on an analogy between the process of natural selection in natural environments. The important difference is that we also employ some heuristics to prune our search tree and do not generate new solutions randomly and indiscriminately. This is an

attempt to combine the best of both techniques, namely pure constructive-iterative and pure search-based techniques. This employs the intelligence of the iterative methods but the hill climbing properties of search-based methods are also present. The increased intelligence helps to reduce the run-time and keeps the search well directed. We now describe the algorithm.

Firstly, the algorithm is provided with an initial solution generated by some constructive heuristics or randomly. This is called the seed solution. The seed solution in our case is an initial schedule which can be made by deterministic scheduling or random scheduling. This gives us the original schedule which can then be modified and refined by evolution.

The elements which constitute the solution have some goodness value. This value denotes how far or how near an element is to its optimum assignment. The optimum assignment results in optimum solution. If the goodness value is high then it is thought to be good for the overall solution too and vice versa for low goodness value. We calculate the goodness values for all the elements of the solution. The elements are then tried to be reallocated using constructive techniques, as near their optimal as possible. The elements which are being allocated are the tasks in our case. Each task's placement in the present schedule can be characterised as to how good or bad it is. Most of the critical tasks should be in good locations i.e., should start at their earliest possible start time.

Next, mutations are made with low probability so that we can jump out of the

local optimum and hopefully traverse the search space to reach better and better solutions. The effect of mutations in our case is to allow any random ready-to-run task to be selected for scheduling at a scheduling step. This will allow us to generate and explore newer solutions in the search space. The search space constitutes all the possible valid schedules for this task graph on the given multiprocessor.

After this step we build the solution again and store it, if it is an improvement over previous ones. We again calculate the goodness values for the new solution and iterate again. This goes on until some stopping criterion is met or for a fixed number of iterations. The generalized algorithm is as follows.

```
Simulated Evolution;
    Begin
        S_0 := GenerateInitialSolution; /* may be constructive or random */
        S := S_0; /* solution S is initialized to S_0 */
        S_best := S_0; /* best solution is initialized to S_0 */
        cp := cp_0; /* control parameter initialized cp_0 */
            For i := 1 to MaxIter do
            Begin
            C_pre := Cost(S);
            FindGoodnes(S); /* evaluate goodness of all elements of
            the solution S */
            S_t := ConstructiveAssignment(S); /* Generate temporary
            solution and store it in S_t*/
            S_t := DoMutation(S_t); /* do mutation with small probability */
            S_t := ReGenerate(S_t); /* make sure S_t is valid a solution */
            Gain := Cost(S_best) - Cost(S_t); /* find gain */
            If (Gain > Random(-cp.0)) then
                S := S_t:
            If (Gain < 0) then
                S_best := S_t;
            If (Cost(S_t) = C_pre) then
                cp := f(cp)
            else
                cp := cp_0;
            End;
        Return(S_best);
    End.
```

The different steps can be enumerated as :

1. Initialization: Providing of seed.

2. Evaluation: Assigning goodness values.

3. Mutation: Introduction of random mutations to the solution.

4. Allocation: Reallocating the selected elements.

5. Regeneration: Generating the complete solution again.

6. Postcomputation: Saving of the best solution and testing of termination conditions.

Initialization

In this step of the algorithm, initial solution is provided. The starting seed can be provided either by the user through a constructive heuristic or can be generated randomly. If the seed provided initially is good then the convergence is fast. If the quality of provided seed is not good then the time required will be more but it will not affect the quality of the final solution. This is because we do major changes initially. In a bad seed more changes are done than in a good seed because the initial goodness of most of the individuals is quite low.

Evaluation

In this step of the algorithm, we compute the goodness values of all the items or elements which make up the whole solution. For good assignments the goodness values are high and low for bad assignments. These values will now be used while making a new schedule once again from the previous ones. It is at this step that we employ the intelligence of heuristic methods and this gives simulated evolution an edge over other search methods.

Mutation

In this step we try to introduce new characteristics in our solution. The mutations are done with low probability to keep the search directed. The mutations are completely arbitrary and will lead to completely different soultions. The probability is kept below 5%.

Allocation

This is a computationally intensive step of the simulated evolution algorithm. Here we do the re-assignment of the elements. This is the constructive part of the algorithm. In this part we can use constructive heuristics which take into account the goodness values of each element.

Regeneration

In this step we make sure that the new solution generated is a valid one. In our case a valid solution is a schedule in which all the tasks (elements) obey the precedence constraints imposed by the communication arcs. This means that all the tasks start only after they get all the required data from their predecessors and there are no overlaps between computations of two distinct tasks. We also find the cost of the new solution which we have just found.

Postcomputation

In this step all the statistics are updated and if the present solution is an improvement over the previous best solution then we save this solution.

The other important thing to check in this step is the control parameter cp_0. If the cost of two consecutive states is the same then this means that we are stuck in a local minima. To escape from this we increase the value of the hill climbing parameter cp. This is done by using a function $f(cp)$ which increases the value of cp. If the gain is greater than zero we always accept the move but if it is not positive then also we can accept the new state depending on value of the control parameter. This is done by generating a random integer between $-cp$ and 0 and accepting the new solution if the gain is greater than this negative random number. As soon as $Cost(S_l) \neq C_{pre}$ we again reset the value of control parameter to cp_0. Thus the basic strategy is to keep cp at a minimum value and increase it only when it is necessary.

Also we check the termination conditions and if they are still not satisfied then we loop back to step "Evaluation".

3.5 Results of simulated evolution compared to other algorithms

This algorithm is much faster then the previous two and gives much better results too. It does not need the careful tuning of parameters as for simulated annealing and large memory space like genetic algorithm. Thus simulated evolution provides good results using little CPU time. This algorithm has given better quality solutions in lesser run time than other such algorithms like Simulated Annealing and Genetic Algorithm for VLSI problems and some typical NP-hard problems like the travelling salesman [17] and for high level synthesis [18]. We expect it to give better solutions in this problem too.

3.6 Conclusions

All the three discussed techniques have been used extensively to solve NP-Hard problems in the field of engineering and computer science. They have been used for computer aided design of VLSI systems for placement of standard cells, routing, circuit partitioning etc, [17], [18], [16], [15]. They have also been used for scheduling and high level synthesis of digital systems.

The choice between them is governed by the case of implementation of each and the run time needed. The Simulated Annealing algorithm is easy to implement. We

do not need to develop any criteria for analyzing a solution and identifying badly placed elements. This means that the problem and the relationships between the elements of the solution need not be studied at all. The solution is only characterised by its cost and we look at it as a whole. The only requirement for us is to find a suitable neighbor function which will help us traverse the search space. The drawback is that more run time is needed to generate acceptable solutions. A lot of experimentation and time is also needed to tune its parameters. If the problem is intractable to analysis and development of heuristics and run time is not a major concern then it is a good algorithm.

The genetic algorithm gives acceptable solutions much more quicker than simulated annealing. The reason is the parallel nature of the search in which more than one solutions are present in each generation. This though requires lot of computer main memory. Here also we do not need to analyze the problem in any detail. The issue instead is how to map the problem to a series of strings as genes and chromosomes. The representation should be such that after applying the genetic operators we can easily get the resulting solution and find its cost. The placement of each and every element of the solution is not assessed and a solution is only characterised by its cost. This algorithm's implementation is more difficult than simulated annealing because it needs the solution to be represented in a particular way. The run time is lesser than simulated annealing for finding acceptable solutions.

The Simulated Evolution is much more intelligent than the previous iterative

methods like simulated annealing and genetic algorithm. It take much lesser time than both of them. The memory requirements are lower than genetic algorithm but more than simulated annealing. This is because though we have only one solution, as in simulated annealing, but we have to store the goodness values of all elements. It requires an in depth study of the problem to identify the badly placed elements and also evaluate the numerical value of each element's placement in the solution. It allows us to control the solution generating process in an intelligent manner. Implementation is difficult than the other methods. It is a must if lower runtime is needed in certain applications. Overall it is an improvement in both the time requirement and the quality of the final solution.

Chapter 4

Literature Review

In this section we present some scheduling heuristics which have been used previously. We give a brief account of the basic principle behind each of them. All of them are constructive heuristics and do not accept a bad solution at any step. One of these refines the generated solution by doing forward and backward iterations. The concept of backward iteration will be explained later. This gives it a better performance as it searches in a larger search space by iterating few times whereas the rest of the algorithms produce a schedule in just one single pass.

There are basically two control approaches for scheduling. The first one is called *Processor Driven* and the second *Computation Driven.* In Processor Driven (PD) approach, the set of ready-to-run tasks called RTR is modified only when the execution of a scheduled task finishes. Thus the scheduling steps are driven by the task finishing times. When this happens all the parallel paths in the graph are given

equal opportunity. This results in a breadth-first expansion. There is a concept of global time here and the starting times of successively scheduled tasks form a non-decreasing sequence.

In Computation Driven (CD) approach the updating of the RTR set is slightly different. As soon as we assign a task on a processor we try to update RTR. If all the predecessors of a task have been assigned (but not necessarily completed execution) then this task is entered in RTR. This results in vertical or depth first expansion. Also the size of RTR will be larger in the CD case than in PD. Now we can select any path for scheduling from amongst some choices and schedule along that path. This gives the notion of critical path and task criticality. At times we will refer to task criticality by task level also.

4.1 List Scheduling

This heuristic was proposed by Graham [5]. In this heuristic the model neglects inter-task communication times so it works with the model $(\Gamma, \rightarrow, \mu)$. As the name suggests we maintain an ordered list of tasks. The criterion for ordering is the task level denoted by $l(T)$. The task level is found from the task graph only and there is no need to look at the task-processor mapping as we are disregarding all the communication edges. So this is a static measure of task criticality. This task level can be computed by traversing the task graph from the exit node up to the entry

node. The value of $l(T)$ for task T is given by Equation 4.1.

$$l(T) = \mu(T) + \begin{cases} 0 & \text{if } succ(T) = \phi \\ Max(l(T_i) : T_i \in succ(T)) & \text{otherwise.} \end{cases} \quad (4.1)$$

In Equation 4.1 we are in effect calculating the maximum distance left after each task to the exit node. Thus for all tasks which have only the exit node as successor the $l(T)$ value is $\mu(T)$. By $succ(T)$ we mean the successors of task T. After finding all the $l(T)$ values the tasks are arranged in a list in non-increasing order of $l(T)$ values. The list is scanned form the beginning and the first task which is also in the RTR is scheduled on a free processor. This task is removed from the list. The RTR set is updated in the PD manner when any scheduled task finishes execution. Because of the PD approach the starting times of successively scheduled tasks form a non-decreasing sequence in time. At each decision step this heuristic schedules the task which has the largest distance left to the exit. This in a way tries to schedule along the critical path of the graph. The performance of list scheduling is 5% away form the optimum in 90% of the cases [19].

We know that the task levels found using the above approach are not very exact as they lack any information about the task processor mapping. This may be quite good in shared memory systems but becomes of crucial importance if inter-task communication times have to be incorporated for DMM systems. The finding of task level for model $(\Gamma, \rightarrow, \mu, C)$, is difficult because we do not know the task-processor mapping prior to scheduling. And the distance or task level of any task

will now depend on the mapping of its related tasks. We can take a pessimistic view and calculate the $l(T)$ values taking into account all the communications. This can be found as shown below.

$$l(T) = \mu(T) + \begin{cases} 0 & \text{if } succ(T) = \phi \\ Max(l(T_i) + c(T, T_i) : T_i \in succ(T)) & \text{otherwise.} \end{cases} \quad (4.2)$$

In Equation 4.2 we account for all the communication. These $l(T)$ values can now be used for scheduling graphs with inter-task communication times considered. The $l(T)$ values so calculated are also inaccurate because we know that if $p(T) = p(T_i)$ then no time is needed for communication and if $r(p(T).p(T_i)) > 1$ then more than $c(T, T_i)$ time is needed for communication. So the definition of task level should be refined more to measure task criticality accurately.

4.2 The ETF Algorithm

The Earliest-Task-First algorithm was proposed by Hwang [4]. This again uses static node priorities. These priorities or $l(T)$ values can be computed by using Equation 4.1. The algorithm uses the CD approach in updating the RTR set. At each step for each task in RTR we find the earliest starting time. The search for processor for each task is exhaustive. When the $est(T)$ for all tasks in RTR have been completed, we select the task having the smallest est value. If there is a tie then we look at

$l(T)$ values and break the tie in favor of the task with higher $l(T)$ figure. Using this method it is possible to schedule a task with higher priority after a task with lower priority. This is because the algorithm looks at $l(T)$ values only when it needs to break a tie else it will just look at $est(T)$ values.

The shortcoming in such approach is that it will schedule a not so important task first if its $est(T)$ figure is sufficiently small. This is a greedy approach and may delay important tasks which lie on the critical path. It may very well happen that a better schedule can be obtained if we delay the tasks which are not critical and leave the resources (processors) for the start of critical tasks.

4.3 The Generalized List Scheduling Algorithm

This algorithm proposed in [20] is a sort of compromise between the previous two methods. It tries to find $l(T)$ values which take into account computation, communications and task-processor assignments. The scheduling decision is based neither on wholly $est(T)$ nor wholly on $l(T)$ but is a combination of the two. The decision function denoted by $d(T)$ is given by Equation 4.3.

$$d(T) = l(T) - est(T) \tag{4.3}$$

The $l(T)$ values are found by scheduling the reverse graph G_r on the multiprocessor $S(P, R)$. The reverse graph as we know already is found by inverting the

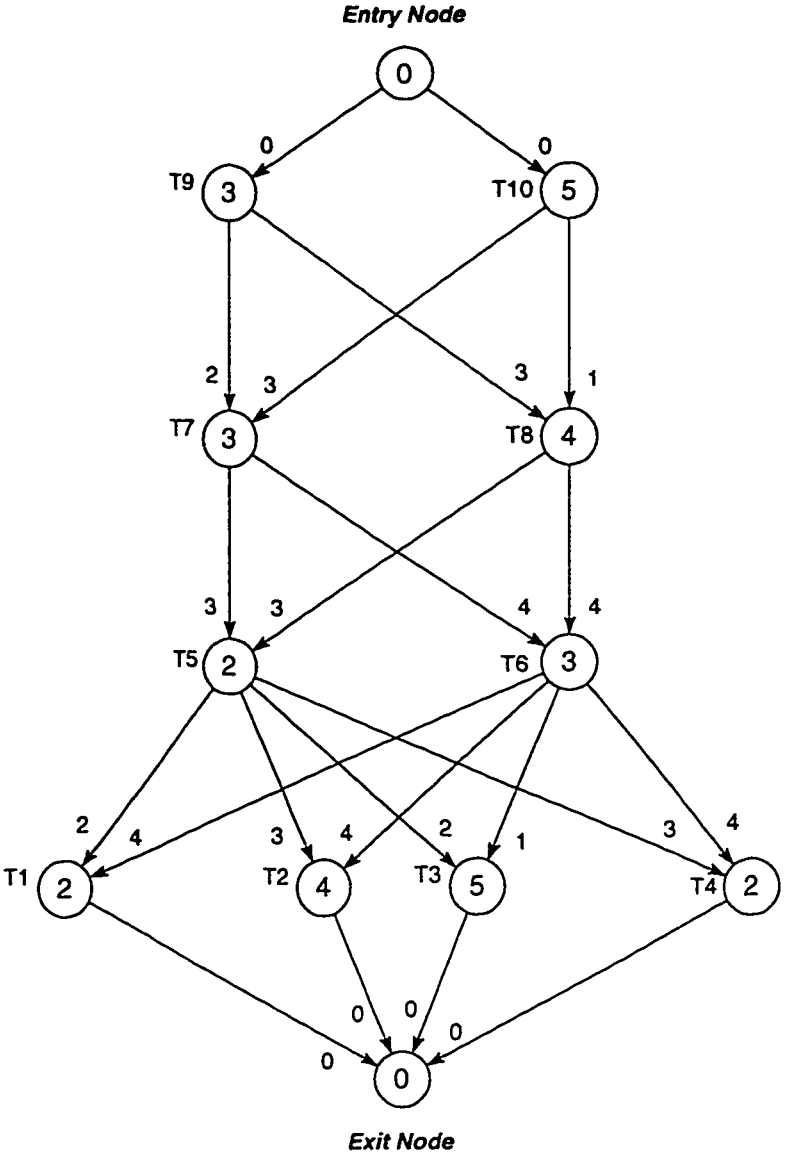


Figure 4.1: The reverse graph of the graph shown in Chapter 2.

direction of all precedence edges and keeping everything else same. This reverse graph is scheduled by using ETF which is a local heuristic. The reverse graph of the graph shown in chapter 2 is shown in Figure 4.1. The $l(T)$ values are just the completion times of the tasks that results from scheduling the reverse graph i.e. $l(T) = ct_{reverse}(T)$. The completion times so utilized are only an approximate indicator of task criticality but they incorporate the most critical effects, along an arbitrary chain of tasks, that are the computations, communications, and network latency.

The decision function $d(T)$ given by Equation 4.3 now uses local ($est(T)$) as well as global ($l(T)$) information. This is just like Graham's list scheduling. The differences are that it takes a different decision function and uses CD approach for updating RTR set. The steps can be enumerated as follows:

1. **Initialization:** Evaluate task levels $l(T)$ for all tasks.

2. **Scheduling:** Until all tasks are scheduled, among the ready tasks, select the task with highest decision function and start it at it's earliest starting time on the processor on which its $est(T)$ was found.

As the decision function given by Equation 4.3 uses both level and earliest start time and the control approach is CD the heuristic is also called $CD/HLETF$ for Computation-Driven/Highest-Level-Earliest-Task-First. This strategy yields solutions that statistically minimize the finish time.

4.4 The Iterative Refinement Scheduling (IRS)

This is an extension to the idea discussed in the previous section [21]. In the previous algorithm first we scheduled the reverse graph G_r, over $S(P, R)$, and obtained the task level values from the finish times of the tasks in the reverse schedule. Then we did one forward pass and used these $l(T)$ values in our new decision function $d(T) = l(T) - est(T)$ to generate a new forward schedule. This process is extended by *alternatively scheduling* G_r *and* G *over system* $S(P, R)$ and each time passing the completion times $ct(T)$ of the tasks as $l(T)$ values for the next iteration in order to search in a space of solutions. This iterative refinement of $CD/HLETF$ gives still better results than doing only one single pass. It is hoped that the passage of $l(T)$ values from forward to reverse iteration will lead in better estimate of task criticality. Actually though, the finishing times of the schedules oscillate in high and low values and can display the following trends:

1. The IRS process converges to the minimum finish time found during the iterations.

2. The IRS converges to a value deviating from the minimum.

3. The IRS process oscillates and does not converge to one value.

4. The IRS does not converge to a value within a fixed number of iterations.

One interesting point to note is that if IRS is started with random values as $l(T)$ for the first iteration then it will quickly recover and generate the same quality solution as was obtained by the deterministic start. But now it requires more number of iterations.

4.5 The CD/ERDETF

This heuristic, Computation Driven/Effective Renmaining Distance Earliest Task First [22], is an improvement over the simple IRS scheduling. It also uses the method of forward and backward iterations. The level values in IRS were simply the finish time of the task in the previous schedule. The level values now are denoted by $erd(T)$, or the effective remaining distance. The $erd(T)$ of a task is a much more refined estimate of its criticality than simple completion times $ct(T)$.

The longest activity path $lap(T)$ approximates the largest sum of computations and communication costs along a path in a given task graph from the entry node. If we add computation time of task $\mu(T)$ and its $lap(T)$ i.e., $erd(T) = lap(T) + \mu(T)$, then this will approximate the remaining computations and communication costs from the starting of T to the exit node in the next iteration. So, this value can be used as a level value or criticality measure for the next iteration.

The level values are thus passed between forward and backward scheduling passes and this results in a local search. At each step of the scheduling process, we schedule

the task which maximises $d(T) = erd(T) - \kappa \times est(T)$. This is exactly the same as IRS but now instead of using $rd(T)$ we use $erd(T)$. This is the best constructive heuristic known to us.

4.6 The Edge Zeroing Algorithm

This algorithm [23] attempts to reduce the partial schedule length at each step by considering the highest cost edge in the task graph. At each step the algorithm schedules the two nodes with heaviest communication edge to the same processor if the partial schedule length does not increase. A list of edges in decreasing order of communication costs is maintained for this purpose. It then removes the first edge from the list and schedules the two incident nodes to the same processor if the partial schedule length does not increase otherwise they are scheduled to two distinct processors. The process is repeated until all nodes are scheduled. The Edge Zeroing algorithm has a time complexity of $O(e(e+v))$, where the number of edges in the graph are e and number of tasks are v. This algorithm tries to minimize communication and fails to exploit the inherent parallelism of a task graph.

4.7 The Dominant Sequence Clustering

This algorithm [24] schedules a task graph on an unbounded number of fully-connected processors. Initially all the tasks are assumed to be mapped to different

processors. After this, for each node we calculate the *tlevel* and *blevel* i.e., the top level and bottom level. The $tlevel(T_i)$ is the largest sum of computation costs and communication costs from an entry task to T_i, excluding $\mu(T_i)$, where $\mu(T_i)$ is the computation time of task T_i. The *blevel* of T_i is the largest sum of computation and communication costs from the start of T_i to an exit node. Now based on a priority value the tasks are clustered by zeroing the communication edges between them. In doing this, two tasks are clustered on the same processor. The critical path of a clustered graph is the longest directed path from entry task to exit task. The dominant sequence, DS is the largest path from an entry task to an exit task in the scheduled DAG, or the directed acyclic graph. Thus DS is the largest computation and communication path taking in consideration the ordering in processors and is the actual length of the schedule at the present clustering. Parallel Time or the schedule length is defined by Equation 4.4.

$$PT = \text{Max}[tlevel(T_i) + blevel(T_i)]. T_i \in \Gamma \tag{4.4}$$

The main idea behind the algorithm is to perform a sequence of edge zeroing steps with the aim of reducing DS at each step. Tasks or nodes of the graph are examined one at a time for clustering. An unexamined node is called free if all of its predecessors have been examined. Priority of any node is the sum of its *tlevel* and *blevel* as given by Equation 4.5.

$$Priority(T) = tlevel(T) + blevel(T) \tag{4.5}$$

The DS nodes have the highest priority values. Initially all nodes are marked as unexamined. Then one free unexamined task with highest priority is taken. The incoming communication edges are then tested for zeroing or not zeroing. If there are free DS nodes they are selected otherwise free $subDS$ (nodes not on DS) nodes are taken for consideration. The whole graph is thus traversed topologically. The criterion for edge zeroing is that the *tlevel* of the highest priority free node should not increase by doing so. By thus compressing the edges, the DS length can be reduced. When an edge is zeroed then the free task is merged to the cluster where its predecessor resides. If last task of this cluster is independent of this free task then a pseudo edge is added directed from the last task to this free task. Updating of *blevel* and *tlevel* values is done only when one of the free tasks' child is being examined. This reduces complexity. At each step a priority list is maintained containing all free tasks of the unexamined part of the graph. This procedure goes on successively until all the nodes have been scheduled.

This algorithm schedules a task only when it is free. This may at times result in the delaying of tasks which lie on the critical path. If the *tlevel* of a task is not reduced by scheduling it to an already been used processor then it will schedule it on a new processor. This may result in the use of more processors than needed. The time complexity of this algorithm is $O((e+v)\log(v))$, where the number of edges in the graph are e and number of tasks are v.

4.8 The Modified Critical Path Algorithm

This algorithm [25] associates an *as-late-as-possible* binding for each task. This $ALAP$ is calculated by traversing the task graph upwards from the exit nodes to the entry nodes and pulling the nodes downwards as much as possible. All nodes are then assigned the latest possible starting times. For every node a list is prepared having the $ALAP$ values of itself and its children nodes sorted in decreasing order. These lists are then sorted lexicographically in increasing order and from this ordering a node list is made. The first node in this list is scheduled to the processor which gives the earliest possible start time. This is done until the list of nodes becomes empty. The time complexity of this algorithm is $O(v^2 \log(v))$, where the number of tasks in graph are v.

4.9 The Mobility Directed Algorithm

In this algorithm [25] another binding called the earliest possible start time or the *as-soon-as-possible* is also associated with each node of the task graph. The $ASAP$ figure for each node is calculated by traversing the task graph downwards from the entry nodes to the exit nodes and by pulling the nodes upwards as much as possible. Mobility is now defined as the difference between $ALAP$ and $ASAP$ values of each node. So $M(T)$ is given by Equation 4.6.

$$M(T) = ALAP(T) - ASAP(T) \tag{4.6}$$

$$M_r(T) = M(T)/\mu(T) \tag{4.7}$$

Relative mobility is computed by Equation 4.7 and is the mobility of the node divided by its computation time. A list is made containing the nodes having minimum relative mobility. A task from this list is selected if it does not have any predecessor in the list. This task is now tried to be inserted in suitable idle time slots in p_1. This slot can be made by shifting the already scheduled nodes on that processor downwards from their $ASAP$ values but earlier than the $ALAP$ values. If this is not possible then p_2 is examined and so on. After this scheduling all the communication edges between this node and previously scheduled nodes on this p are changed to zero. If this node T_i is scheduled before T_j then a pseudo-edge is added directed from T_i to T_j, if T_i is scheduled after T_j then the opposite happens. This is done to prevent deadlock. Relative mobilities of all nodes are now recomputed. This goes on until all the nodes have been scheduled. The time complexity of this algorithm is $O(v^3)$, where the number of tasks in graph are v.

4.10 The Dynamic Critical Path Scheduling

In this algorithm [26] the tasks are assigned dynamic priorities depending on the current schedule. This algorithm does not take a fixed multiprocessor configuration but rather it tries to minimize the number of processors as much as possible. When scheduling a task on a processor it does not simply minimize the start time but also estimates as to what effect it will produce on that node's successors or children with regard to their starting times. When considering a processor for scheduling a task only those processors are chosen which have successors or predecessors of this task scheduled on them.

We now look at this algorithm in a little more detail. The *Absolute Earliest Starting Time* of a node T_i on processor p is denoted by $AEST(T_i, p)$ and is the time this node can start its execution on p after receiving all its data from its parent nodes. This varies from processor to processor, depending on the time it takes to deliver messages from the last parent to the specific processor. Similarly the *Absolute Latest Starting Time* of a node T_i on processor p is denoted by $ALST(T_i, p)$. This in effect is the time beyond which if the node is delayed then the schedule will get delayed too. This is found by traversing the graph in the reverse direction i.e., starting from the exit node. The $AEST$ of exit node is the schedule length minus the computation time of the exit node. The *Dynamic Critical Path Length* or $DCPL$ is calculated from Equation 4.8.

$$DPCL = \text{Max}[AEST(T_i, p(T_i)) + \mu(T_i)] : T_i \in \Gamma \tag{4.8}$$

Those nodes which have equal $AEST$ and $ALST$ values are identified to be on the critical path. These are the nodes which fall on the longest path from any entry node to exit node. The node to be scheduled is on the DCP and is the one having no unscheduled parent nodes on the DCP. The schedule at each p is checked, and if there exists a vacant slot between tasks k and $k+1$ scheduled at this p, there is a possibility of scheduling this task on this processor. Otherwise if there is no such empty slot then we can create it by pushing node k to its $AEST$ value and node $k+1$ to its $ALST$ value. This slot should be at least equal to the computation time of the node being considered for scheduling. Afterwards the $AEST$ and $ALST$ values of the affected nodes are updated. This strategy does not select the processor that gives the earliest starting time. Instead it tries to minimize the function $AEST(T_i, p_j) + AEST(T_c, p_j)$, where T_c is that child node which has smallest difference between its $AEST$ and $ALST$ values and is called the critical child. This is to forestall the condition that T_c's critical children can not find slots on p_j and thus may be increase the overall schedule length. Not all processors are tested for scheduling, only those who have either parent or child node of T_i scheduled on them, are tried. Finally all the nodes are made to start at their $AEST$ values. The time complexity of the algorithm is $O(v^3)$, where v is the number of tasks in

the graph.

4.11 The Dynamic Level Scheduling

In this algorithm [27], a value called the *Dynamic Level* is assigned to each node at every scheduling step. This is computed by using two quantities. The first is called the static level, $SL(T_i)$ and is the maximum sum of computation costs along a path from T_i to an exit node. The second quantity called the data available time, $DA(T_i, p_j)$ is the earliest time at which all the data transfers from its parent nodes to p_j can be completed. The dynamic level is then defined as given by Equation 4.9.

$$DL(T_i, p_j) = SL(T_i) - DA(T_i, p_j) \tag{4.9}$$

All the nodes whose parents have all been scheduled are called ready nodes. The DL values for each ready node are calculated for each available processor. Now if $DL(T_i, p_j)$ is the largest value from amongst all the calculated values then T_i is scheduled to p_j, the DL values are recomputed for the present setup and this goes on till all the nodes have been scheduled. This algorithm does not assign priorities based on the critical path. Exhaustive pair matching of nodes to processors is done to find the highest priority node.

4.12 Task Duplication Scheduling

The Duplication Scheduling Heuristic [28], tries to reduce communication by executing some tasks on more than one processors. This redundant execution can help in providing the precedence data to some critical tasks and thus reduce the finish time of the schedule built using task duplication. The main steps of the algorithm are enumerated below.

1. Make a list of processors (PL) which have the predecessors of critical (also called candidate task) tasks (CT) on them.

2. For each processor p in (PL) do the following.

3. Find the idle time slot, which is the difference between the finish time of last task scheduled on p and earliest starting time of a candidate task on p.

4. Find the earliest starting time of the dominant predecessor (D_{CT}): this is the predecessor whose data arrives latest for the candidate task. If $est(D_{CT}, p)$ is within the idle time slot of p then duplicate this task in the idle slot.

5. If the task is accommodated in the idle slot then apply steps 2 and 3 for this processor.

6. If the slot is overflowed and the starting time of CT is not lowered then restore original situation.

7. Repeat from step 2 for the next dominant predecessor of the CT.

8. Schedule CT on the processor which gives minimum starting time.

9. Repeat from step 1 until all tasks are scheduled.

The task duplication algorithm has a time complexity of $O(v^4)$, where the number of tasks in graph are v. The time complexity is high and the management of duplicate messages is difficult. Another new method using the duplication approach was given in [29].

This Chapter presented a review of the determinsitic and iterative scheduling heuristics which have been in the literature recently. The time complexities and other salient features of these methods were also described.

Chapter 5

The Arc-Addition Method

The concept of mobility is an attractive one. Task criticality based on mobility has been investigated and we have tried developing some heuristics using mobility. The tasks having zero mobility in a schedule are on the critical chain. If the critical chain is scheduled in a better manner then we may be able to get shorter schedule lengths. We now describe this method and how to find mobility.

We will now describe the mobility finding process in more detail. The process starts by accepting a previous schedule made by a computationally cheap deterministic algorithm. This schedule is now analyzed and we try to find those tasks which are contributing to the length of the schedule. These are the tasks which are said to lie on the critical chain. We call a chain of tasks critical, if we lengthen the overall schedule by delaying any task on this chain. So in effect we can say that the tasks lying on the critical chain are the most important contributors to the overall

schedule length. If some how we can place the tasks on the present critical chain in a better way then we may end up with a schedule with a lesser finish time.

Thus, our problem is to select these tasks by finding an accurate criticality measure for a given graph and a given schedule. We have a previous schedule found in the previous iteration. This schedule gives the information about start and finish times of each task and their processor assignment. Let us consider a small task graph of 10 tasks and we try to schedule it on four fully connected processors. The task graph is shown in Figure 5.1.

Now let us see a schedule of this task graph on four fully connected processors. This is shown in Figure 5.2(a). This schedule has length of 22 time units. Now we try to find the critical chain for this schedule. The method of finding the critical chain is by trying to do a reverse scheduling. In this we start from the exit nodes and build up a schedule honoring the previous processor allocation. This way we again find the start times and finish times of the tasks. This is shown in figure 5.2(b).

We represent the forward schedule start time by $st_f(T)$ and the pulled-down schedule start time by $st_{pulled}(T)$. Mobility is then defined by the difference in these times as shown in Equation 5.1.

$$M(T) = st_{pulled}(T) - st_f(T) \tag{5.1}$$

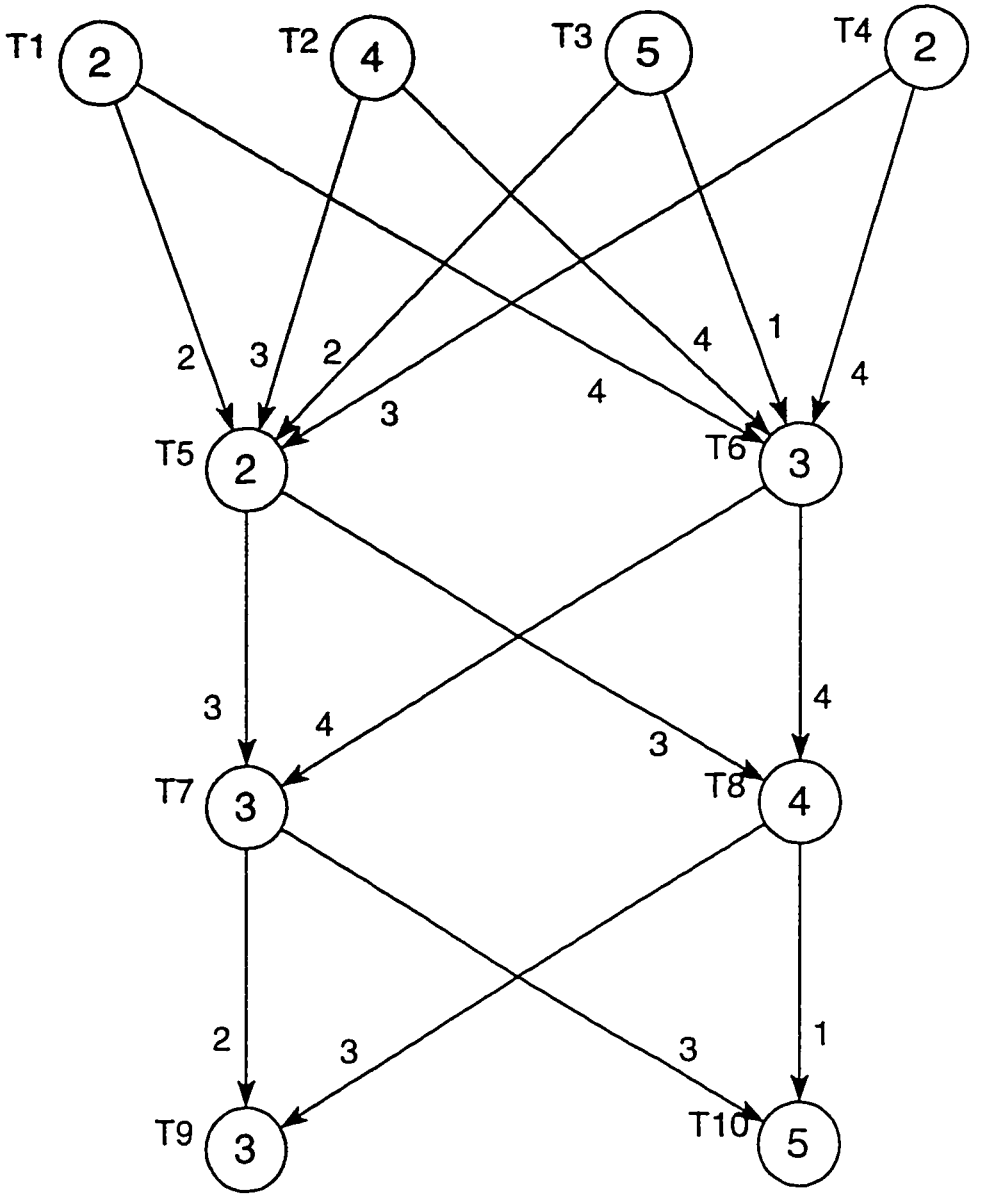


Figure 5.1: The task graph for the example.

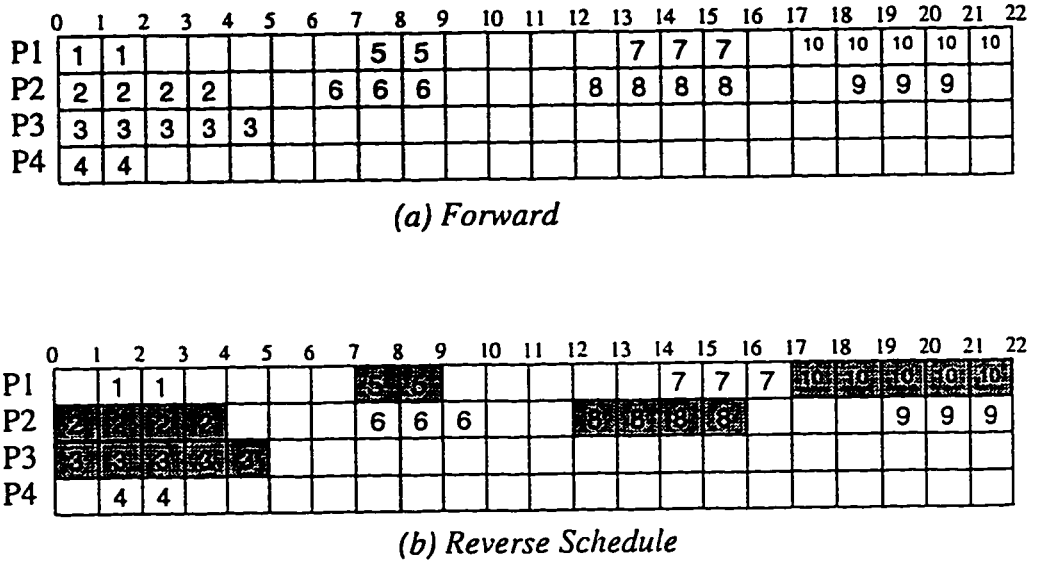


Figure 5.2: The forward and reverse schedules for the example.

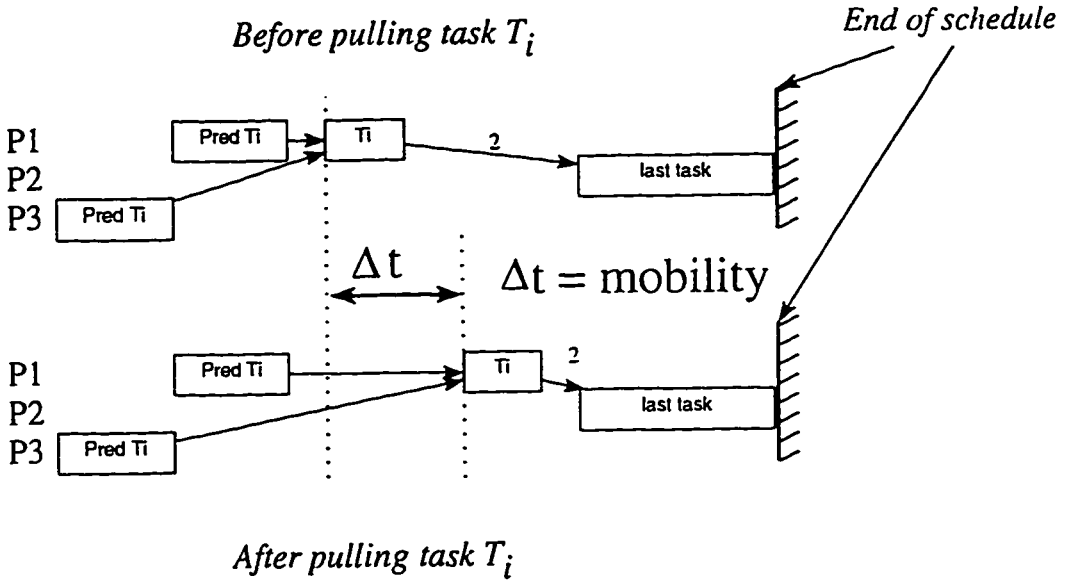


Figure 5.3: The operation of finding mobility of task T_i.

As $st_{pulled}(T) \geq st_f(T)$ the mobility is always non-negative. If the mobility is zero it means that this particular task starts at the same time in both the schedules. Generally for other tasks $st_{pulled}(T)$ will be higher because the pulled-down schedule is made by trying to pull down all the tasks as much a possible while honoring the previous processor allocation and the previous total schedule length. Thus both the forward and pulled-down schedules have the same overall length. So, by identifying all the tasks with zero mobility we can find the tasks which are contributing to the present schedule length. We can therefore say that all the tasks having zero mobility are on the critical chain. In our example all the tasks having zero mobility are shown shaded. There can be forks and joins on this chain as is evident from the example.

The technique of finding the mobility is also shown in Figure 5.3. The technique is simple and just requires pulling all the tasks towards the End of Schedule mark as much as possible. This pulling down or making the tasks start at their latest starting time is done such that the length of the schedule remains same. If this pulling down of a task is possible then it is not on a critical chain. If the pulling down is not possible then the task starts at the same time in both the forward and reverse schedules and its mobility is zero. Otherwise we have definite non-zero amount of mobility. So for each task we have a figure of criticality with respect to a given schedule.

If for a task $M(T)$ is zero then it has no slack in the present schedule and if we hope to do better then we should try to start it earlier than it's starting time in the

present schedule. If a task has a non-zero $M(T)$ then the amount of slack for this task is $M(T)$ and it can be safely delayed by this amount and chance given to more critical tasks. The higher the $M(T)$ figure for a task the more we can safely delay it giving precedence to tasks with lesser $M(T)$.

5.1 Observations

While doing the iterative refinement we tried keeping a tab on the mobility of each task in the resulting schedules. If we ran 10 iterations of iterative refinement using $d(T) = rd(T) - \kappa \times est(T)$ as our decision function we observed a particular phenomenon.

In each resulting schedule (we will get 20 schedules in 10 iterations of iterative refinement) we recorded which tasks were having zero mobility i.e., $M(T) = 0$. We represented this figure by *zero mobility confidence* or $zmc(T)$. Now this figure was unusually high for some tasks. The upper limit for $zmc(T)$ is 20 and we found some tasks having a very high figure i.e., more than 15 to up till 20. Similarly for some tasks it was very low and even zero.

This indicated that the tasks having high $zmc(T)$ values were on the critical chain in majority of the schedules as formed by the iterative refinement process. As the iterations continued these tasks mostly lied on the zero-mobility chain or the critical chain. On the other hand, for the tasks having low values of $zmc(T)$ we

can conclude that they almost always were not lying on the critical chain in the 20 schedules generated by the iterative process.

5.2 Conclusions

From the evidence in the previous section we can conclude that $zmc(T)$ figure of a task is an estimate of its importance in the schedules which can be generated by the iterative process. The iterative process searches in a solution space and in this solution space these tasks are important or non-important depending on their $zmc(T)$ figures.

This information can now be used in the second phase of the iterative refinement process which uses the information generated in the first phase of the process. We try to make some changes in the schedules keeping in mind the $zmc(T)$ values. Our aim will be to modify the scheduling process in such a way that we get better solutions using $zmc(T)$.

5.3 The Arc-Addition Iterative Refinement

The arc-addition iterative refinement uses the concept of pseudo arcs. The main algorithm still uses the same technique as the usual iterative refinement. The decision function is still kept the same. By using pseudo arcs we can achieve a lot of control over the scheduling process. The pseudo arcs can be added between any two

nodes of the task graph. These two tasks after the addition of the pseudo arc, are connected by a communication edge of a very large weight. The arc can only be added so that it does not introduce cycles in the graph. After addition of this pseudo arc between tasks T_1 and T_2 i.e., $T_1 \rightarrow T_2$ we achieve the following two conditions.

1. A precedence relationship is established between tasks T_1 and T_2.

2. T_1 and T_2 will be scheduled on the same processor now.

A precedence is established because the pseudo arc is treated exactly like a normal one. We can add a pseudo arc between related tasks or even between tasks which were previously unrelated to each other. The two tasks connected by a pseudo arc will always be on the same processor. This happens because the weight of the pseudo arc is so large that it will enforce both of them together. We know that at each step we schedule the task having largest value of $d(T) = rd(T) - 100 \times est(T, p^*)$. The processor on which the earliest starting time was found is denoted by p^*. If $p(T_1) \neq p(T_2)$ then $est(T_2)$ will be very large on $p(T_2)$. This will happen because a very large amount of time will be required to send data from $p(T_1)$ to $p(T_2)$; this time is equal to $r(p(T_1), p(T_2)) \times c(T_1, T_2)$. As the weight $c(T_1, T_2)$, is very large $est(T_2)$ will be large and the decision function will be a very large negative number. The scheduler will therefore schedule the two tasks such that $p(T_1) = p(T_2)$ and $r(p(T_1), p(T_2)) = 0$. If both the tasks connected by the pseudo-arc are on the same processor then in effect it does not disturb the original topology of the computation

graph. So the schedule is a valid one for the original graph also and this is what we desire.

The arc-addition process hopes to enforce some conditions on the schedule and then within these conditions it allows the iterative refinement process to continue. The addition of arcs can later be done within the framework of simulated evolution and we hope that this will lead to a good scheduling technique. The evolution process can associate a performance measure for each pseudo arc and then try to mix and match between the available set of individual arcs. This addition of more than one arc can result in a sort of chains through the graph and these pseudo arcs will enforce the tasks of these chains on one processor in a definite precedence order. There can be more than one chain and all the tasks of a given chain will be scheduled on the same processor.

5.4 Criteria for Arc Addition

The crucial point now is how to add these arcs by looking at the $zmc(T)$ values collected in the first phase of the iterative refinement. For this purpose we designed a work bench where we can add arcs interactively and then allow the iterative process to run with this arc. We can also suggest more than one arc for a given run. The arcs can be suggested by looking at the $zmc(T)$ values of the tasks. This setup allows us to view the original task graph as well as the minimum schedule found

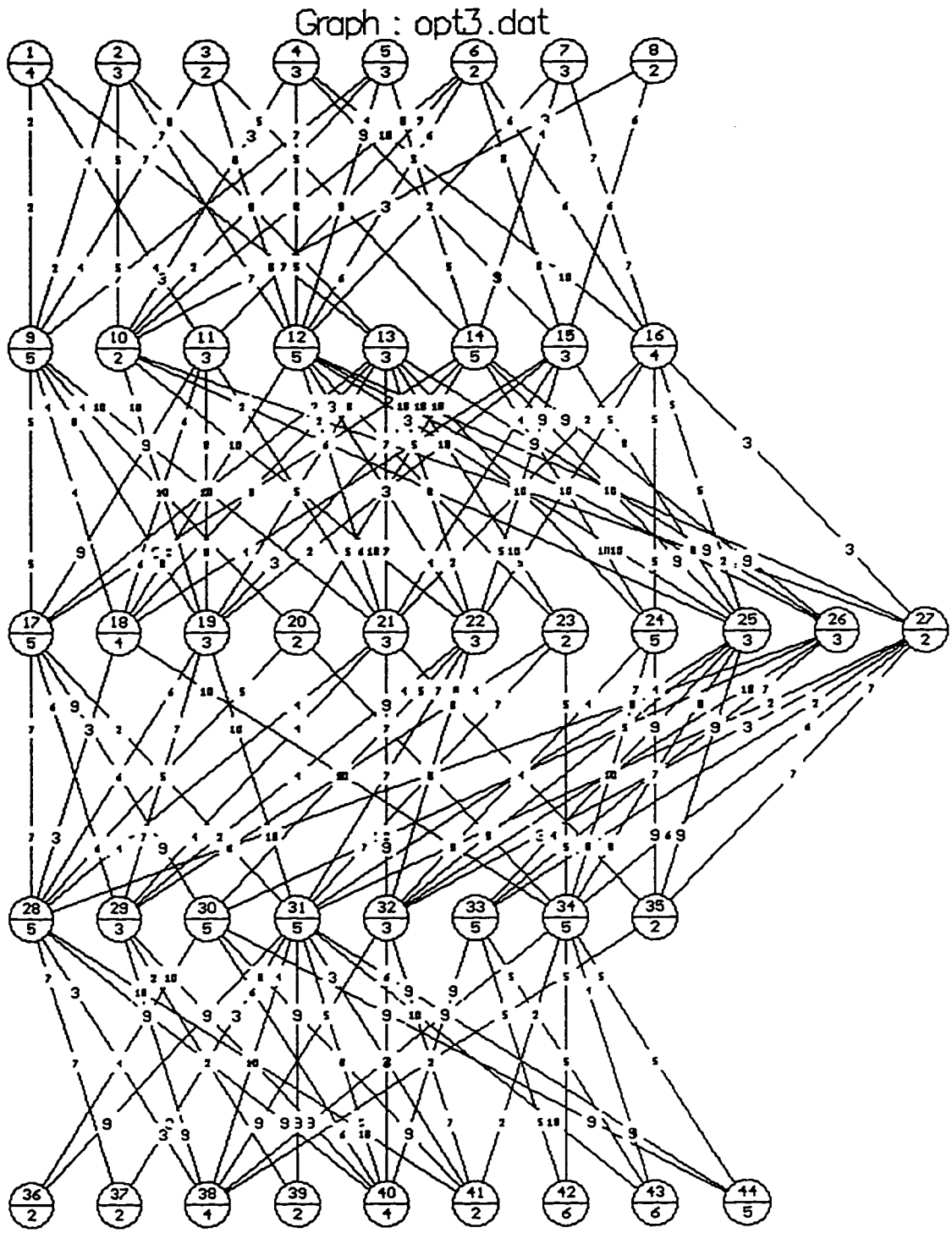


Figure 5.4: A task graph as displayed on the 640x480 VGA display by the program.

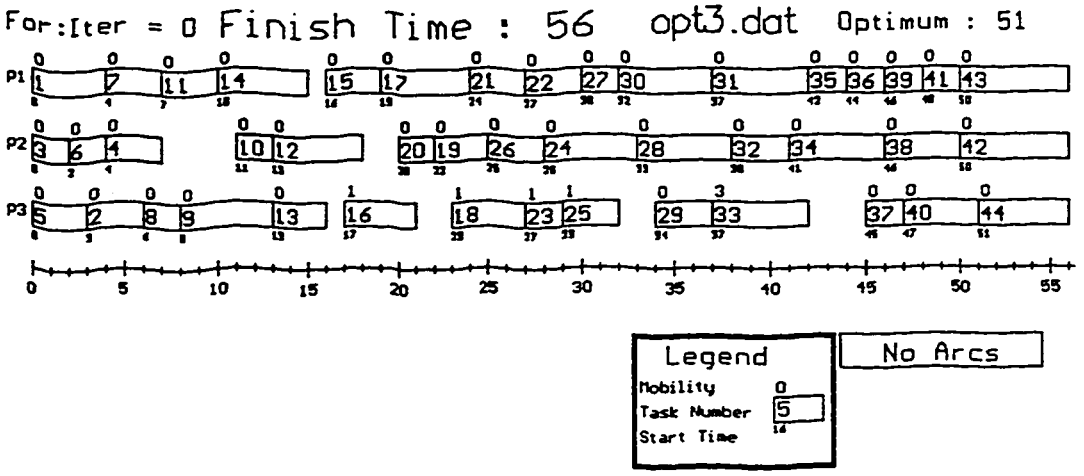


Figure 5.5: The Gantt chart of the previous task graph as displayed on the 640x480 VGA display.

during the refinement process. A task graph and a schedule is shown in Figures 5.4 and 5.5. The Gantt chart or the schedule also displays the starting time, completion time, mobility, and computation time of each task. This process of experimentation was very helpful and yielded a set of rules on which arcs to add.

The first important thing observed was that some graphs gave their best finish time always in the forward schedule while others always gave their best finish time in the reverse one. The reverse schedule is generated by the backward iteration of the IRS process. The backward iteration is exactly the same as forward and the only difference is that it uses the reverse graph. In the reverse graph all the precedence arc directions are reversed and the entry tasks become the exit tasks and vice versa. The schedule generated for a reverse graph is also valid for its forward graph.

This character of each graph was consistent and a graph giving its best in a

forward run would keep doing so irrespective of the iterations we do on it. The same will happen for a graph giving its best in reverse schedule. This categorizes our graphs in two groups; ones which give their best solution in forward schedules and the others which do so in reverse schedules. The arc-addition operations are found to be different for each group. The addition of arcs is further sub-divided into two groups. We can add arcs between entry or exit tasks or we can add arcs between tasks which are embedded deeper within the graph.

It was observed that we should concentrate on tasks which are in the first level for task graph which are of the forward type and on bottom level tasks for the graphs which give their best on the reverse schedule. Our intention basically is to identify one high $zmc(T)$ task in the entry or exit level (depending on the graph) and then try different pairings with another important task of the same level. This way we try to overlap computation with communication for critical tasks critical successor in the next level. This gives us our first rule for finding pseudo-arcs. The steps are as follows:

Rule 1

1. Select the highest $zmc(T)$ task in entry level. call it T_1.
2. Select the highest $zmc(T)$ task in next level, call it T_3, such that T_3 is a successor of T_1, i.e., $T_3 \in succ(T_1)$.
3. Now for each T_2 such that $T_2 \in pred(T_3)$, we have an arc $T_2 \rightarrow T_1$.

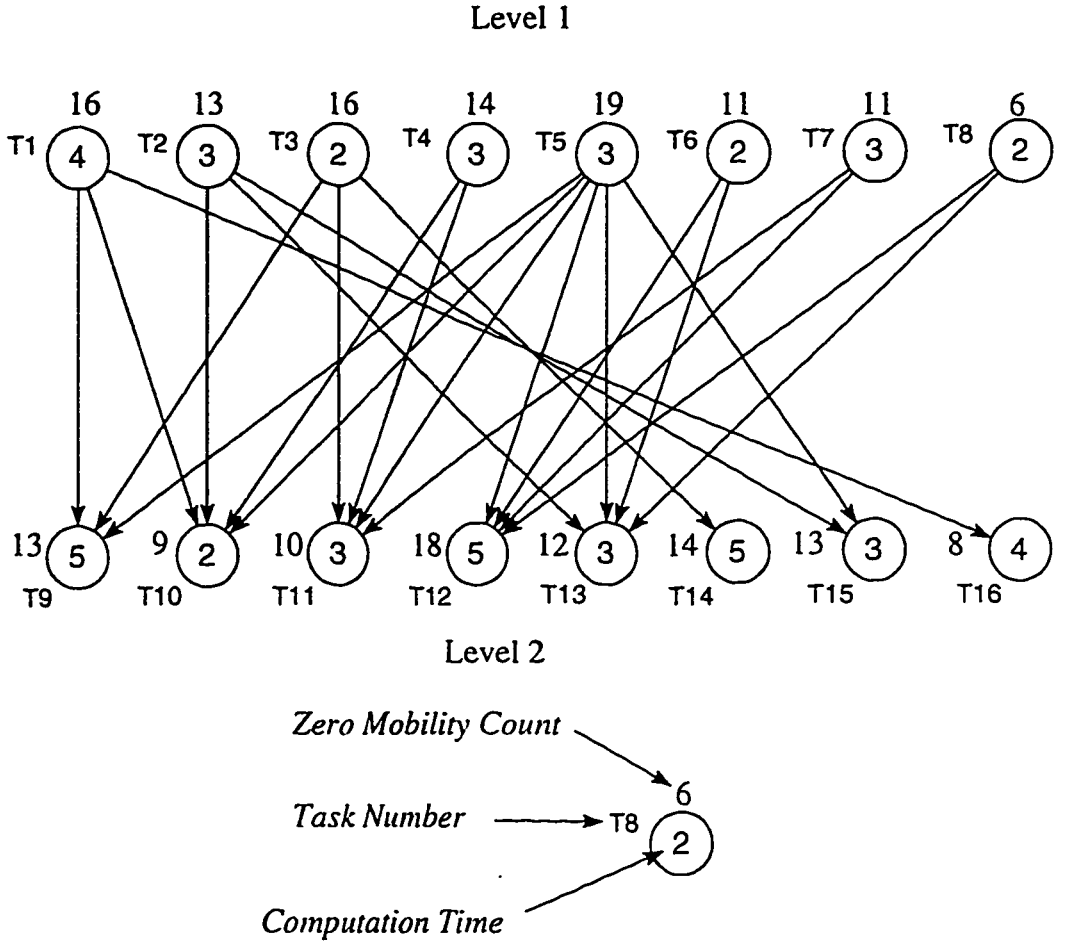


Figure 5.6: An actual example of arc addition process.

4. Also we have arcs $T_1 \rightarrow T_2$. This increases the search space by doubling the number of potential arcs.

The above method is used for the graphs which give their best finish time in forward iterations. For graphs giving best results in the reverse iterations we apply the same steps but now we do so for the last level and the one-before-last level tasks. The addition of these pseudo arcs was found to give a much better performance.

An example is shown in Figure 5.6. This is taken from an actual task graph after 10 runs of iterative refinement. This graph yielded its best result in the forward iteration so we will work with the first level. The $zmc(T)$ values of each task are given with each task. In the first level we have T_5 as the task having highest $zmc(T)$ value. Now we look at the successors of T_5. Its successors are $T_9, T_{10}, T_{11}, T_{12}, T_{13}$ and T_{15}. The most critical amongst them is T_{12}. The predecessors of T_{12} excluding T_5 are T_6, T_7 and T_8. This gives us the following set of arcs : $T_5 \rightarrow T_6$, $T_5 \rightarrow T_7$, $T_5 \rightarrow T_8$, $T_6 \rightarrow T_5$, $T_7 \rightarrow T_5$ and $T_8 \rightarrow T_5$.

Rule 1 was concerned with only the entry or exit tasks of the graph. But it was found that sometimes we need to add pseudo arcs even between tasks which are embedded deep in the graph. This gave us our Rule 2.

Rule 2

1. Starting from level 2 search a task having the highest $zmc(T)$ value at that level. Call it T_1.

2. Search a task from amongst the successors of T_1, called T_2 such that its $zmc(T)$ figure is highest and its processor in the minimum schedule is different from the processor of T_1. Thus we have $T_2 \in succ(T_1)$ and $p(T_2) \neq p(T_1)$ in the minimum schedule.

3. The pseudo arc found is $T_1 \rightarrow T_2$.

4. Repeat the above steps until a minimum number of arcs are found.

The above rule is based on the observation that two crucial and related tasks were on different processors in the minimum schedule found by the first phase. So it would be better if we somehow can place them on the same processor. This is what is achieved by adding such a pseudo arc. The number of such arcs to be found is a parameter that can be varied. We tried different combinations. We found that adding a number equal to 5% of number of tasks is good enough. If we add more arcs we have to do much more iterations and the quality of solution may not necessarily increase by this. If we add too few, we may risk not finding the best this method can yield.

5.5 Implementation

The implementation of this method was divided into two parts. The first part had two phases. The first phase was the generation of $zmc(T)$ values, getting the

minimum schedule from the iterative refinement process, the categorization of the graph as forward or reverse. The second phase of the first part is concerned with the application of the rules and generating a set of prospective pseudo arcs. For the iterative process we used the best heuristic known to us. This is the CD/ERDETF or the Computation-Driven/ Estimated-Remaining-Distance Earliest-Task-First. The decision function is $d(T) = erd(T) - 100 \times est(T)$. The use of $erd(T)$ as a measure of task criticality rather than using simple completion times as level values, gives much better results [22]. The estimated remaining distance is in a way a sharper estimate of task criticality. Thus our heuristic uses CD/ERDETF as its base. The second part of the new technique is again sub-divided into two parts. This is dictated by the simulated evolution methodology. According to simulated evolution we need to have a measure of the performance of each move. The moves here are the addition of pseudo arcs. In the first sub-part of the second part we try to find the performance of each arc individually. This is done by adding each arc one at a time and allowing the iterative process of CD/ERDETF to continue for a 10 iterations. The minimum finish time found during this time is the performance of this arc.

After we know the performance of each arc we sort them in the decreasing order of their performance. The arcs giving the better finish times have a higher chance (probability) of selection than other not so good arcs. Now our aim is to build a set of pseudo arcs which can give us a better solution than all the solutions found so far. The performance of individual arcs we already know. The process now is to add

more than one arc and trying iterative refinement. If we get a better schedule we accept the new arc unconditionally. If the process gives us a worse solution we may accept it probabilistically according to simulated evolution concept. We may also add mutated pseudo arcs. These arcs can be between any two tasks of the graph. After adding all the pseudo arcs for an iterative refinement run we check if we have introduced any cycles in the graph. If there is a cycle we do not proceed with 10 iterations of iterative refinement but try to find new settings.

This is a sort of a branch and bound strategy in simulated evolution environment. We go on adding arcs if the solutions resulting are better but if it yields bad results we remove that arc. The whole set up is shown in pseudo-code form in Figure 5.7.

We briefly explain the steps of the algorithm. The process begins by getting the initial set of level values from the procedure **GetInitialLevels**. This procedure takes as input the reverse graph G_r and the multiprocessor $S(P,R)$. It does one pass of ETF scheduling and returns the set of completion time values $ct(T)$ as task level set L. We then do 10 runs of iterative refinement using L for the first run. The procedure **IterativeRefinement** takes three parameters namely, G, the graph: $S(P,R)$, the multiprocessor; and L the set of initial levels. This procedure returns its best solution in S_{best}. It also returns the zero mobility confidence set ZMC and the type of the graph. The type tells us whether we got the minimum in forward iteration or the reverse iteration.

The set of possible arcs is found by using the information provided by the pre-

```
Program Evolution-Based Arc-Addition Scheduling;
  Begin
    n_arcs := 0;
    L := GetInitialLevels(G_r, S(P, R)); /* using constructive ETF on G_r */
    S_best := IterativeRefinement(G, S(P, R), L); /*return best
    solution so far, mobility-confidence or ZMC and graph type */
    S := S_best;
    ArcSet := FindArcs(G, ZMC, graphtype);
    FindPerformance(ArcSet);
    cp_0 := Cost(S_best)/100 ; /* initial value of control parameter */
    cp := cp_0; /* control parameter initialized to cp_0 */
    For i := 1 to MaxIter do
        Begin
        C_pre := Cost(S);
    100: If (50 > random(100) ) then
            Begin
            n_arcs := 1;
            Arc[n_arcs] := ProbSelect(ArcSet); /* add pseudo arc */
            End;
    110: If (50 > random(100) ) then
            Begin
            n_arcs := n_arcs + 1;
            Arc[n_arcs] := ProbSelect(ArcSet); /* add pseudo arc */
            End;
         cycle = CheckCycle(Arc);
         If ( cycle ) then
            Begin
            n_arcs := n_arcs - 1; /* remove last pseudo arc */
            Goto 110;
            End;
    120: If (20 > random(100) ) then
            Begin /* do mutation */
            n_arcs := n_arcs + 1;
            Arc[n_arcs] := RandomSelect(G);
            cycle = CheckCycle(Arc);
            If ( cycle ) then
               Begin
               n_arcs := n_arcs - 1; /* remove mutated pseudo arc */
               Goto 120;
               End;
            End; /* mutation done */
         S_t := IterativeRefinementArc(G, S(P, R), Arc, L);
         Gain := Cost(S_best) - Cost(S_t); /* find gain */
         If (Gain < 0) then
         S_best := S_t;
         If (Cost(S_t) = C_pre) then
         cp := f(cp) /* f(cp) = 1.2 × cp */
         else
         cp := cp_0;
         If (Gain > Random(-cp, 0)) then
         S := S_t; /* accept solution */
         Goto 110 /* accept all pseudo arcs */
         else
         Goto 100; /* remove all pseudo arcs */
         End;
      Return(S_best);
  End.
```

Figure 5.7: Pseudo-code for Evolution Based Arc-Addition Scheduling

vious procedure. This is done by the procedure **FindArcs**. It takes as input the graph G, the ZMC values and the *graphtype*. It returns the set of possible arcs in *ArcSet*.

The performance of each arc is found by the procedure **FindPerformance**. This procedure takes as its parameter the set of all the arcs found i.e., *ArcSet*. It then uses the procedure **IterativeRefinementArc** to find the performance of each arc individually.

The initial value of the control parameter is set at 1% of the best solution cost. Now the simulated evolution phase of the program begins. The array *Arc* is used to store the set of arcs to be used in the iterative process. We add first arc with a 50% probability. The selection is done by the procedure **ProbSelect**. It takes as input the set of arcs *ArcSet* and returns a selected arc. This arc is selected based on its performance. This means that an arc having a good individual performance has a higher chance of selection. This adds the first pseudo arc. Again with a probability of 50% we choose the second arc similarly. While this second addition we disregard the first added arc so that it may not get selected again.

After adding these two arcs we check if they have introduced any cycles in the graph. If there are any cycles we make a new selection. If there are no cycles we proceed to the mutation phase. Here we randomly choose any two tasks from the graph G and add an arc between them. We take care not to add an arc which we have already added. Again we check for the introduction of any cycles and if there

are no cycles we proceed ahead. If there are any cycles then we remove the last added arc.

The iterative refinement process is then applied using the procedure **IterativeRefinementArc**. It takes as input the added set of arcs and returns the minimum found in the 10 iterations. If the new cost is equal to the previous cost we increase the control parameter. If the new cost is lesser than the previous best cost we accept the new solution as well as the present set of pseudo arcs. The addition process now continues builds from the present set of pseudo arcs in Arc. If the solution is not accepted we remove all the pseudo arcs and begin again. The best solution is returned after we have exhausted $MaxIter$ iterations.

5.6 Empirical Testing of Arc Addition Scheduling

Now we will present the results of empirical testing. The results are compared with other constructive techniques also. The comparisons are done with the single iteration CD/ETF algorithm and with $CD/ERDETF$. We have already talked about the CD/ETF algorithm.

The Computation-Driven/Effective-Remaining-Distance-Earliest-Task-First, $CD/ERDETF$ [22], is a constructive-iterative technique. This is just like the previous iterative refinement technique which was discussed previously. This uses as

priority measure an estimate of remaining distance called $erd(T)$ and the decision function is as given by Equation 5.2.

$$d(T) = erd(T) - 100 \times est(T) \tag{5.2}$$

The set ERD of $erd(T)$ values is passed from forward to reverse iterations and the iterative process refines the $erd(T)$ or effective-remaining-distance and the process converges generally in under 20 iterations. This method was discussed more detail in Chapter 4.

5.7 Task-Graph Generation and Performance Testing

A random graph generator RGG, was used to generate task graphs with different communication and computation properties. The sizes of the graphs vary from 180 tasks to 200 tasks. The computation times of all graphs vary from $\mu_l = 1$ to $\mu_h = 19$ with $\mu_{average} = 10$. The communication times $c(T.T')$ were varied from 0 to 59 units. The communication is allowed to vary between C_{low} and C_{high}. The factor α refers to the ratio of average communication $C_{average}$, divided by average computation $\mu_{average}$. This is given by Equation 5.3. This characteristic of the task graph takes values from 0 to 3 in steps of 0.5.

$$\alpha = \frac{C_{average}}{\mu_{average}} \tag{5.3}$$

Another characteristic of a task graph is its degree of parallelism. This signifies how many tasks are present at each level. The degree of parallelism is denoted by β and is a measure of the concurrency of tasks in the graph with respect to the number of processors available in the multiprocessor. The total number of levels in a graph are denoted by N_l. So the number of tasks at each level are n_{tasks}/N_l. The number of processors are given by p. Then β is given by Equation 5.4. The random graph generator takes two values for number of levels and the number of levels in each graph is varied between the high and low limits, the high limit or $Levels_h$ and low limit or $Levels_l$. So $N_l = (Levels_h + Levels_l)/2$.

$$\beta = \frac{n_{tasks}}{N_l.p} \tag{5.4}$$

The parallelism degree of the task graphs is taken for values $(0.5, 1, 2, 2.5, 3, 4, 5)$. The Tables 5.7 and 5.7 show the values set for task graph size ranging from $180-200$ tasks and computation ranging from $1-19$ units.

We considered two processor topologies for testing. They are fully-connected or FC, and hypercube or HC. In each case we considered 8 processors. For each value of α and β we generated 30 graphs using the RGG, the random graph generator. In all there are 49 types of graphs and there are 30 graphs of each type.

α	C_{low}	C_{high}
0	0	0
0.5	1	9
1.0	1	19
1.5	1	29
2.0	1	39
2.5	1	49
3.0	1	59

Table 5.1: Variation of C_{low} and C_{high} for different values of α

β	$Levels_l$	$Levels_h$
0.5	45	50
1.0	22	25
2.0	11	13
2.5	9	10
3.0	8	8
4.0	6	6
5.0	4	5

Table 5.2: Variation of $Levels_l$ and $Levels_h$ for different values of β

The results show the average percentage deviation of each of the three methods from ω_{best}. The three methods are:

1. CD/ETF

2. CD/ERDETF

3. Arc Addition

The ω_{best} figure is the minimum finish time found by any of the three methods for a given graph. If the finish time of the heuristic is w_h the percentage deviation is $(w_h - w_{best}) \times 100/w_{best}$. At each value of α and β we have 30 graphs and therefore we average the percentage deviation to get a single point. The number of individual arcs suggested ranges between 10 to 15. Each arc is given 10 iterations. To display the time required to obtain the best solution by Arc-Addition technique we show the arc number at which the best solution occurred. From the results we see that this technique is much better than simple CD/ERDETF.

It was found from the results that the algorithm failed in the accumulation of more than one arc. As soon as more than one arc was added we observed either a deterioration or the same performance as with one arc. Thus in a way simulated evolution methodology of building up a set of arcs which give a better performance than their individual performance, couldn't succeed.

The one bright point was that performance was much better than simple CD/ERDETF without arc-addition. From this we conclude that the addition of one pseudo arc by

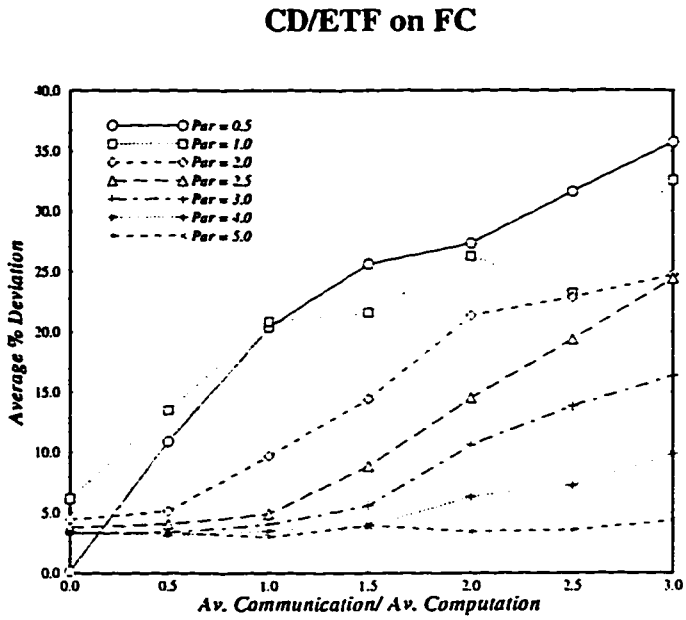


Figure 5.8: Deviation from best known solution of CD/ETF on FC topology.

itself is enough for guiding the heuristic in the correct direction. If we try to enforce more chaining we obstruct the heuristic and get worse solutions. So this method is a sort of deterministic technique.

5.8 Analysis of Results

The performance of CD/ETF is shown in Figures 5.8 and 5.9 for the fully-connected and hypercube topologies respectively. They show a deterioration in performance with increasing communication and an enhancement in performance with increasing parallelism. This behavior is consistent with the nature of EST scheduling. The analysis which was done in the previous chapter for CD/ETF applies here as well. The worst performance deviates from ω_{best} by up to 35% for the fully-connected and

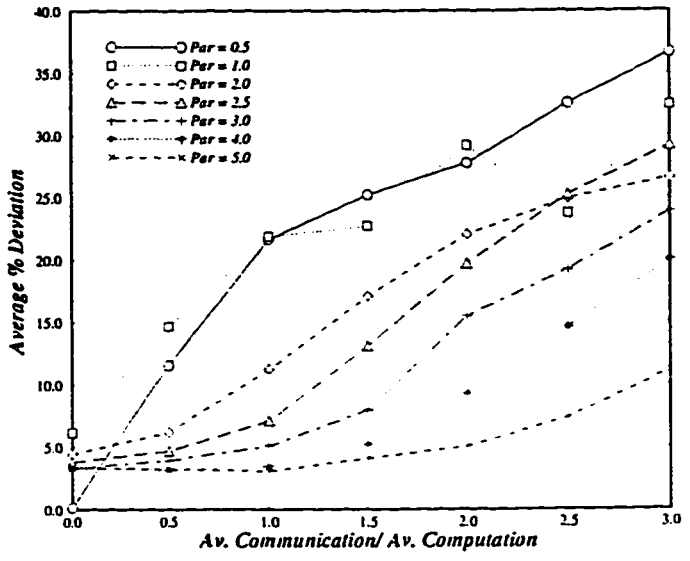


Figure 5.9: Deviation from best known solution of CD/ETF on HC topology.

37% for the hypercube case.

The performance of CD/ERDETF is good. This is shown in Figures 5.10 and 5.11. The maximum performance deviation from ω_{best} is up to 4.3% for the fully-connected topology and 4.7% for the hypercube. The analysis done in the previous chapter applies here as well.

The Arc-Addition method gives excellent results. The results are always better than the other two methods in every case. The addition of a single pseudo arc is proving to be enough to give the performance an edge of nearly 5% over CD/ERDETF. The performance of arc addition on fully-connected topology is shown in Figure 5.12. The arc-addition method always finds the minimum in an overwhelmingly large number of cases. For the parallelism degrees of $\beta = 0.5, 1.0, 3.0, 4.0, 5.0$

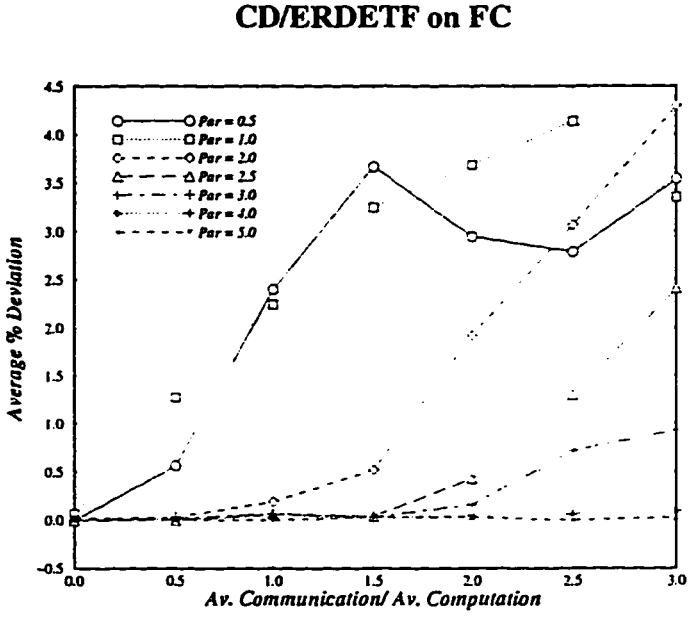


Figure 5.10: Deviation from best known solution of CD/ERDETF on FC topology.

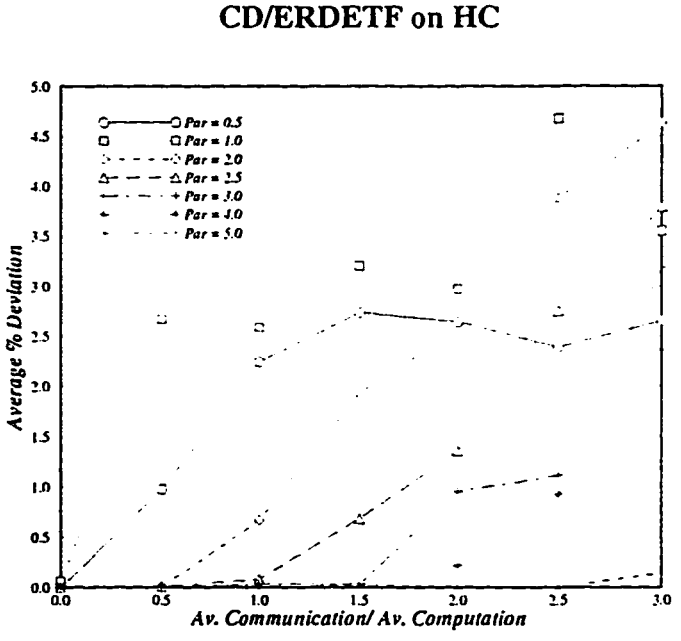


Figure 5.11: Deviation from best known solution of CD/ERDETF on HC topology.

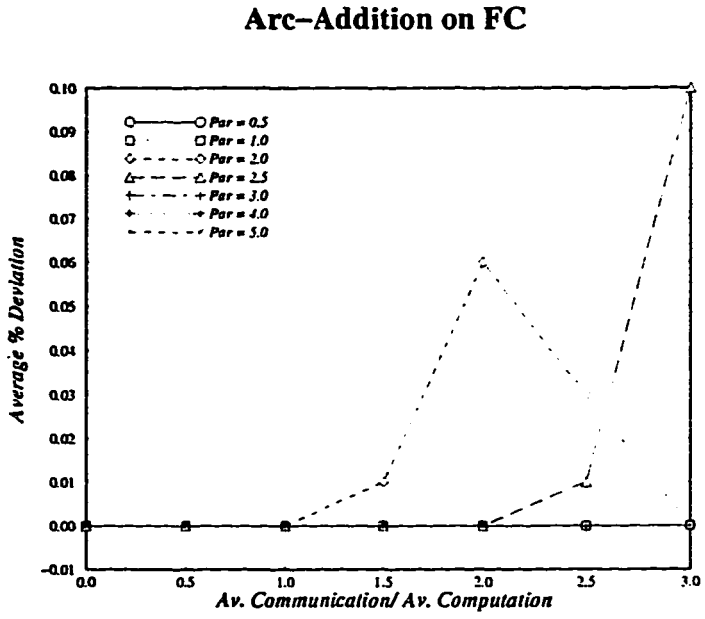


Figure 5.12: Deviation from best known solution of Arc-Addition scheduling on FC topology.

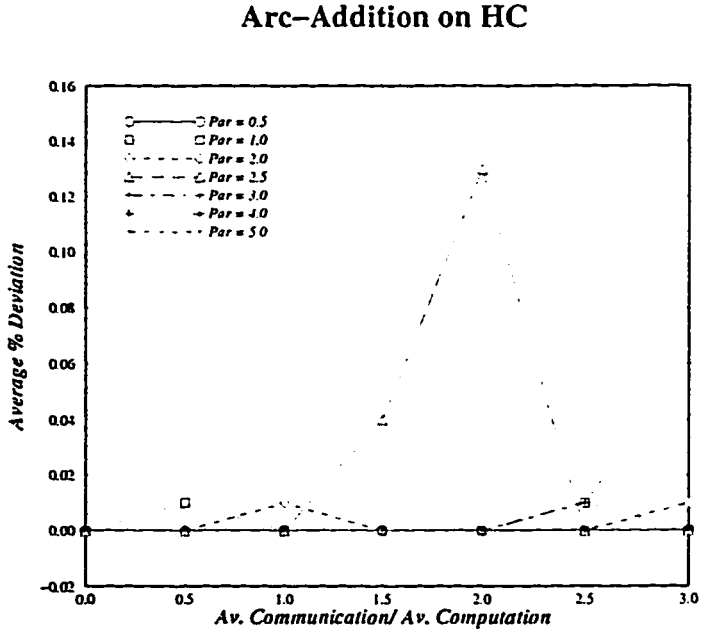


Figure 5.13: Deviation from best known solution of Arc-Addition scheduling on HC topology.

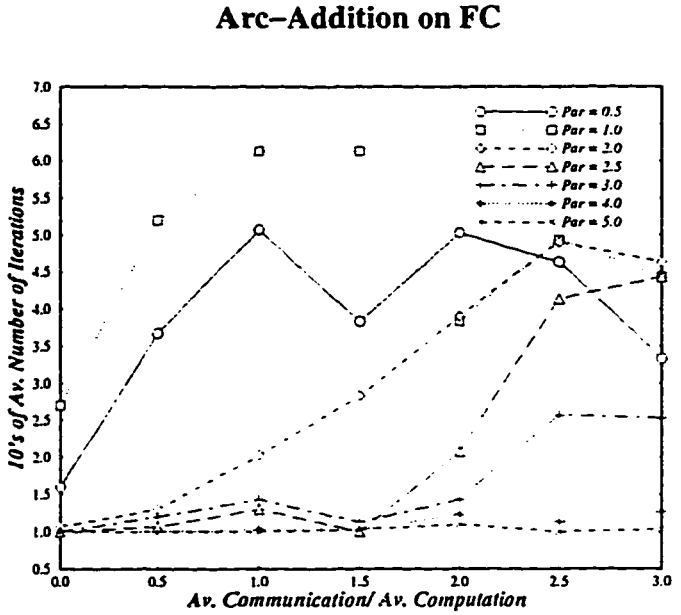


Figure 5.14: Average number of iterations required by Arc-Addition scheduling on FC topology.

the deviation at all values of communication is 0 from ω_{best}. The maximum deviation is at $\beta = 2.5$ and $\alpha = 3.0$ and is 0.1% from ω_{best}.

The results are similar for the hypercube topology. The only effect again is that which happens due to increase in communication. From Figure 5.13 we can see that the worst performance is obtained at $\beta = 2.5$ and $\alpha = 2.0$ and is 0.13% away from ω_{best}.

The number of iterations are always 10 for a given arc. Before the arc addition process begins a number of pseudo arcs are selected for trial according to the given rules. In Figures 5.14 and 5.15 we show the arc number at which the minimum was found. It is implied that for each pseudo arc we do 10 iterations of iterative refinement. From Figure 5.14 we see that most of the search is done at low values

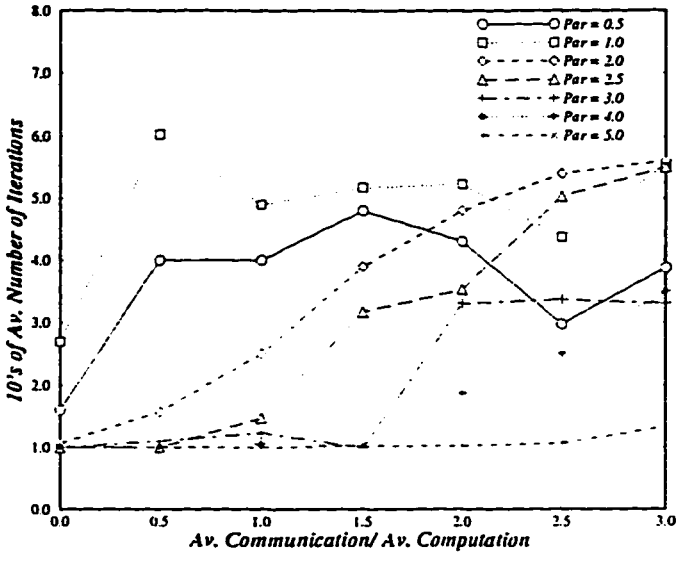


Figure 5.15: Average number of iterations required by Arc-Addition scheduling on HC topology.

of parallelism. The worst case is obtained when we have $\beta = 1.0$ and $\alpha = 1.0$. In this case we need to try on an average 6.2 arcs for finding the best solution. The effort decreases as parallelism increases. Nearly same situation exists for the hypercube topology. This is shown in Figure 5.15.

The Tables 5.8 and 5.8 summarise the overall results across all values of parallelism and communications. We had 1470 graphs for each topology. Between any two heuristics there are three possibilities. They are : heuristic A performing equal to heuristic B, heuristic A performing better than heuristic B and heuristic B performing better than heuristic A. We give the percentage number of cases where each of the above three situations are present in the second column. In the third column we give the average percentage deviation in finish time for each of the three

Heuristic	*% of Cases*	*% Deviation*
CD/ETF = CD/ERDETF	1.70	0.00
CD/ETF better than CD/ERDETF	0.06	0.23
CD/ERDETF better than CD/ETF	98.24	11.18
CD/ETF = Arc Addition	1.70	0.00
CD/ETF better than Arc Addition	0.00	0.00
Arc Addition better than CD/ETF	98.30	12.40
CD/ERDETF = Arc Addition	65.10	0.00
CD/ERDETF better than Arc Addition	0.41	1.05
Arc Addition better than CD/ERDETF	34.49	2.99

Table 5.3: Pairwise comparison between heuristics on fully-connected topology

Heuristic	*% of Cases*	*% Deviation*
CD/ETF = CD/ERDETF	1.70	0.00
CD/ETF better than CD/ERDETF	0.00	0.00
CD/ERDETF better than CD/ETF	98.30	12.88
CD/ETF = Arc Addition	1.70	0.00
CD/ETF better than Arc Addition	0.00	0.00
Arc Addition better than CD/ETF	98.30	14.47
CD/ERDETF = Arc Addition	58.78	0.00
CD/ERDETF better than Arc Addition	0.68	0.85
Arc Addition better than CD/ERDETF	40.54	3.27

Table 5.4: Pairwise comparison between heuristics on hypercube topology

cases. The percentage deviation in this case is not based on ω_{best} but is based on the minimum from the two heuristics being compared.

Chapter 6

The Mobility Based Evolution Scheduling

In this chapter we will present our evolution based scheduling technique. Several different methods were tried before getting the final version of the technique. From experience it was noticed that it is best to work at the level of one schedule only. By this we mean that we should not try to mix the task level values resulting from different schedules. It is best to take a schedule and then go on refining the task level values with the evolution concept.

We know that the iterative refinement scheduling uses a decision function of the type $d(T) = l(T) - est(T)$, where $d(T)$ is the value of decision function for task T. $l(T)$ is its task level and $est(T)$ is its earliest starting time on a particular processor. At each scheduling step we schedule the task with highest $d(T)$ from amongst the

ready tasks. The RTR set is updated using the CD or computation driven approach. The decision function $d(T)$, uses two criterions, one is the accurate local criterion, the $est(T)$, and the other is a global criterion and is the estimated value of task criticality, the $l(T)$ or task level.

Given a set of $l(T)$ values and a task graph we will generate a particular schedule each time. So if the decision function is fixed and the control approach is fixed (CD) then a schedule can be specified from a set of $l(T)$ values. This means that we can construct different schedules depending on different sets of $l(T)$ values. The problem is how to accurately judge between different tasks and estimate their criticality so that we can assign them correct $l(T)$ values. The next section describes one approach that was tried to estimate a good value for $l(T)$.

6.1 History Based Estimation of Task Levels

In this approach we first do several runs of the iterative refinement scheduling. In this phase we schedule the graph G and the reverse graph G_r on $S(P, R)$ one after the other. In each iteration we pass the the completion time $ct(T)$ as the task level for the next iteration. This phase is the statistics collection phase of the technique. We store what $l(T)$ values did a task take and what finish time that generated. So for each task there is a profile or history information. Now from the study of the profile we try to estimate a new value for each task. This profile also tells us which

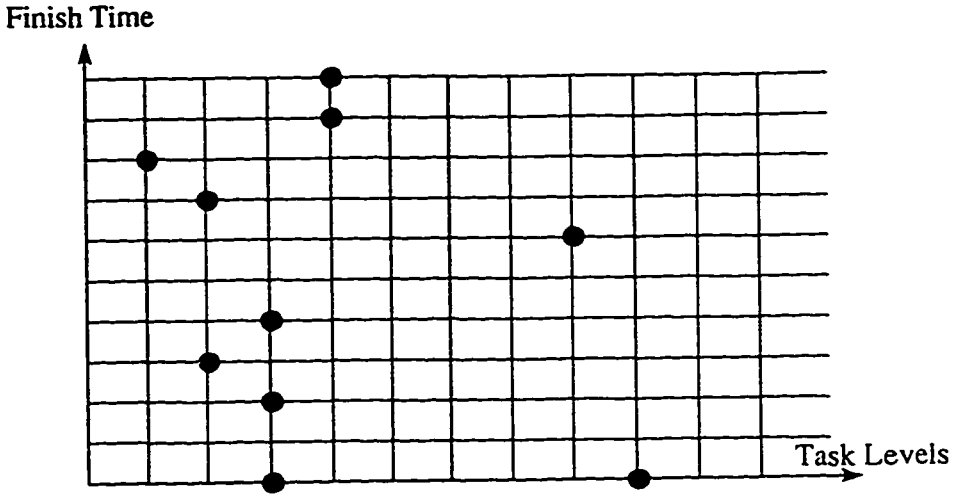
tasks are critical and which are not. For critical tasks there is a larger probability that a change in its $l(T)$ value will reflect as a change on the entire schedule because it is very important that a critical task start at the correct time.

After the statistics collection phase is over we enter the next phase. In this phase we only do forward iterations only with graph G. For each iteration we estimate new task level values for the tasks by using the profile information. As each scheduling is completed the statistics are again updated to reflect the performance of the latest run. Statistics for the last 10 iterations are kept. Therefore for each task we can get profile charts as shown in Figure 6.1.

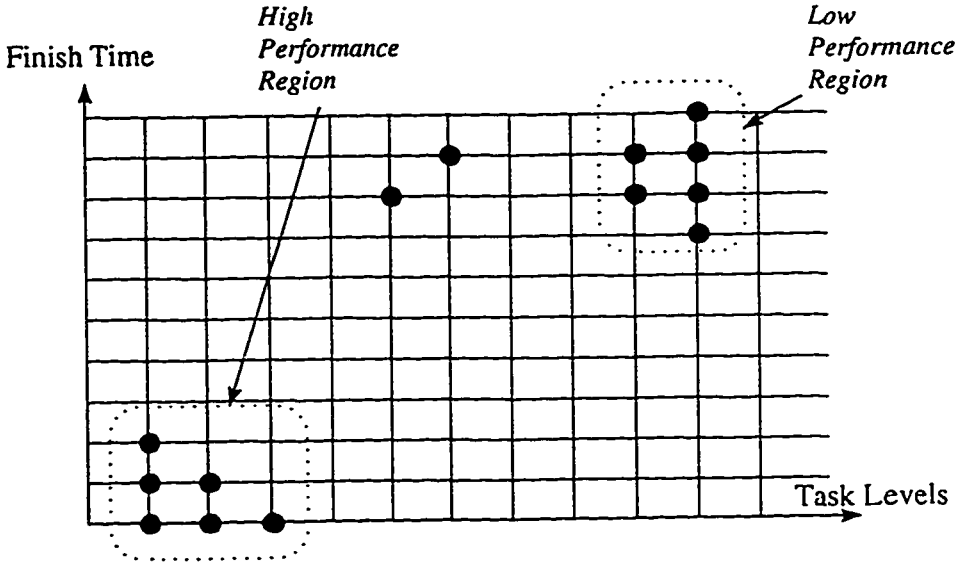
The results from the above technique were not encouraging. In fact the results were worse than simple Iterative Refinement. The conclusion we can draw from this is that each schedule is an independent schedule and mixing task levels from different schedules only confuses the heuristic. So, instead of using information from different schedules we should try to refine a single schedule only.

6.2 History Based Estimation of Task Orders

The above technique was also applied to estimate the task order or $to(T)$. At each scheduling step we assign one task. The order in which they are selected for assignment is an indicator of the priority given to this task by the heuristic. In this case we keep the information about the orders and not the task levels. The



Profile of Task Levels taken by a non-critical task



Profile of Task Levels taken by a critical task

Figure 6.1: Profile charts for a non-critical and a critical task.

statistics collection phase is the same as above with the difference that now we store the information telling us what was the performance of a schedule when a particular task took a particular order.

In the next phase of this technique we only do forward iterations. The assigning decisions are now based on the history profile of orders. Thus for each task we can decide an order and schedule it at that specific order. The best order is chosen from the history profile for each task. Conflicts may arise because two or more tasks may require the same order or a task to be scheduled at a specific order may not be in the RTR yet. The second type of conflict is rare though. If the conflict is of the first type then we break the tie using the old decision function, $d(T) = l(T) - est(T)$, where $d(T)$ is the value of decision function for task T, $l(T)$ is its task level and $est(T)$ is its earliest starting time on a particular processor. The $l(T)$ values now used are the ones which gave the least finish time during the statistics collection phase of the heuristic. On the other hand, if the conflict is of the second type then we delay our not-ready-task until it comes in the RTR. Statistics or the profile information is again kept updated for the last 10 iterations.

The results here were even worse than the above technique. The reason is not using the $est(t,p)$ information at all. We only use the global information provided by the order profiles. This information is not accurate and conflicts arise. The conflicts arise because the same order may have been taken by two (or more) tasks in two different schedules and both may have resulted in good schedules. It may

also happen that in one schedule we go along a chain of tasks deeper into the graph and it gives us good finish time. But in another schedule we may go more in a breadth first manner and that may again give us good finish time. Now one of the deeper embedded task may demand an earlier order but in the current schedule in may not be in RTR yet. So the information obtained from differing schedules is not accurate.

6.3 Mobility Based Evolution Scheduling ($MBES$)

The basic concept of simulated evolution is to manipulate a given solution in such a way that the solution quality becomes better with increased iterations. Simulated evolution searches for items which are badly placed in the present solution. This selection of items is the most important part of the algorithm. If this selection is good then we can identify items (task mapping to processors in our case) which are contributing to the length (finish time) of the present solution (the schedule).

So, after identifying the tasks which are contributing to the present schedule length we can hope to start them in a more favorable position during the making of the next schedule. This, in our case, is done by increasing their chance of getting a processor as soon as they become ready to run. If we can not refine our present solution then we move to another point in the search space. This as we already know is done by the action of mutations.

The critical task identification is based on the mobility which we evaluate as explained in the last chapter. This evaluation provides as a set M of mobilities after analyzing a previous schedule. This value of mobility will then be utilized for scheduling the tasks in a better manner. The incorporation of mobility in the decision function is described next.

The earlier decision function was $d(T) = l(T) - est(T)$, where $d(T)$ is the value of decision function for task T, $l(T)$ is its task level and $est(T)$ is its earliest starting time on a particular processor. The $est(T)$ here is the local parameter and $l(T)$ is the parameter which helps us distinguish a critical task from a non-critical task. The new decision function is given in equation 6.1.

$$d(T) = l(T) - \kappa \times est(T) \tag{6.1}$$

The factor κ with $est(T)$ is used to penalize the idle time slots more. This in general gives better finish times. The reason is that we give more weight to the early starting of the task and if there is a large idle time in starting a task then we will start a task which lessens this idle slot. The local parameter is thus given more weight. The value of κ is around 100.

If the idle time slots generated are nearly the same then we distinguish the critical task and the priority or task level comes into play. If we wish to increase the priority and hence the selection chances for a critical task then we should try to

increase its $l(T)$ value. The same effect of increasing may be obtained by reducing the $l(T)$ values of all non-critical tasks. As now we try to refine a given schedule we do not need to do backward or reverse scheduling. In a starting solution we have the first set of $l(T)$ values and the resulting schedule. We analyze the schedule and find the mobility of each task. The critical tasks in this schedule will have zero mobility. To move to another point in the search space we should have a new set of $l(T)$ values. If the new set of $l(T)$ values gives us a better result we accept the result always, but if the new $l(T)$ values give a worse result then we may accept this probabilistically using the simulated evolution concept. The new values suggested for the next iteration are as given in equation 6.2.

$$l_{i+1}(T) = l_i(T) - M_i(T) \tag{6.2}$$

The values $l_{i+1}(T)$ have been suggested by heuristic means for the $i+1$th iteration. The values which generated the present schedule are denoted by $l_i(T)$ and $M_i(T)$ are the mobilities of tasks in the present schedule. This can lead us to a local minima. So using the simulated evolution technique we do mutations with a low probability. In mutation any random value is assigned to $l(T)$ for the tasks which are being mutated. We randomly select a certain percentage of tasks for mutation. This is a crucial parameter and its correct value can only be gained by trial and error. Different values between $1 - 10\%$ were tried. The best results are obtained when we mutate 3% of the total tasks.

After generating a new schedule from the mutated $l(T)$ values we check the new finish time obtained. The entire set of task level values is referred to by L. If the finish time is the best obtained so far we accept the present set L unconditionally. On the other hand if the finish time is not the minimum seen so far we update L probabilistically depending on how far it is from the minimum. By accepting the new set L we mean that from now on the task level values which are in L will be used to generate the schedules. If the new schedule length is slightly larger than the minimum seen so far it has more chance of acceptance. If we accept a set L which is giving a finish time much larger than the minimum then we will loose a good working point and therefore acceptance chance in such a case is very low.

This acceptance is governed by the value of the control parameter cp. The control parameter is initially kept at a small value so only small losses are accepted. If the initial value is set at 1% of the minimum finish time then any schedule having finish time more than 1% will never be accepted. But if the algorithm gets stuck in a local minima and we get same finish time in two consecutive runs we increase the hill climbing factor or the control parameter. It is increased by 20% when this happens. As soon as the situation changes the control parameter is again reset to the initial low value. The initial low value of the control parameter is about 1% of the finish time of the seed solution. We now give the overall algorithm.

```
Program Mobility Based Evolution Scheduling;
  Begin
    L := GetInitialLevels(G_r, S(P,R));
    /* using constructive ETF on G_r */
    S_0 := MakeSchedule(L, G, S(P,R));
    S := S_0; /* solution S is initialized to S_0 */
    S_best := S_0; /* best solution is initialized to S_0 */
    cp_0 := Cost(S_0)/100 ; /* initial value of control parameter */
    cp := cp_0; /* control parameter initialized to cp_0 */
        For i := 1 to MaxIter do
        Begin
        C_pre := Cost(S);
        M := FindMobility(S, G, S(P,R)); /* evaluate mobility
        of all tasks in the solution S */
        L_t := Suggest(L, M); /* generate new levels form
        L and M and store in L_t */
        L_t := DoMutation(L_t); /* do mutation on 3% tasks */
        S_t := MakeSchedule(L_t, G, S(P,R));
        /* S_T is temporary solution */
        Gain := Cost(S_best) - Cost(S_t); /* find gain */
        If (Gain > Random(-cp, 0)) then
        L := L_t;
        If (Gain < 0) then
        S_best := S_t;
        If (Cost(S_t) = C_prc) then
        cp := f(cp) /* f(cp) = 1.2 × cp */
        else
        cp := cp_0;
        End;
    Return(S_best);
  End.
```

Procedure GetInitialLevels($G_r, S(P,R)$);
Begin
$RTR := [\,]$;
For i := 1 to n_{tasks} **do** /* n_{tasks} is total number of tasks in G */
If ($T.npred = 0$) **then** /* $T.npred$ = Number of predecessor of T */
$RTR := RTR + [T]$;
While($RTR \neq [\,]$) **do**;
Begin
Find($est(T,p)$) for all tasks in RTR;
$d(T_{assigned}) = Min_{T_i \in RTR}\{est(T,p)\}$;
Schedule($T_{assigned}$ on $p(T_{assigned})$ at $est(T_{assigned},p)$) :
$ct(T_{assigned}) = est(T_{assigned},p) + \mu(T_{assigned})$;
$RTR := RTR$ - $[T_{assigned}]$;
For all $T_i \in T_{assigned}.succ$ **do**
If(all predecessors of T_i scheduled) **then**
$RTR := RTR + [T]$;
End;
Return(CT); /* Return set of completion times as initial levels */
End;

Procedure MakeSchedule($L, G, S(P,R)$):
Begin
$RTR := [\,]$;
For i := 1 to n_{tasks} **do** /* n_tasks is total number of tasks */
If ($T.npred = 0$) **then** /* $T.npred$ = Number of predecessor of T */
$RTR := RTR + [T]$;
While($RTR \neq [\,]$) **do**;
Begin
Find($est(T,p)$) for all tasks in RTR:
$d(T_{assigned}) = Max_{T_i \in RTR}\{l(T) - 100 \times est(T,p)\}$;
Schedule($T_{assigned}$ on $p(T_{assigned})$ at $est(T_{assigned},p)$);
$ct(T_{assigned}) = est(T_{assigned},p) + \mu(T_{assigned})$
$RTR := RTR$ - $[T_{assigned}]$;
For all $T_i \in T_{assigned}.succ$ **do**
If(all predecessors of T_i scheduled) **then**
$RTR := RTR + [T]$;
End;
Return(S); /* Return the complete schedule as solution S */
End;

```
Function Cost(S);
    Begin
        Cost := ct(T_1);
        For i := 2 to n_tasks do
        If (Cost < ct(T_i) ) then
        Cost := ct(T_i);
        Return(Cost); /* Return ct of the latest finishin task */
    End;


Procedure FindMobility(S, G, S(P, R));
    Begin
        For i := 1 to n_tasks do
        Begin
            Find(st_pulled(T_i))
            M(T_i) := st_pulled(T_i) - st(T_i); /* st(T_i) is from S */
        End;
        Return(M); /* return the set of mobilities */
    End;


Procedure Suggest(L, M);
    Begin
        For i := 1 to n_tasks do
        Begin
            l_t(T_i) := l(T_i) - M(T_i);
        End;
        Return(L_t); /* return the set of temporary levels */
    End;
```

Procedure DoMutation(L_t);
Begin
$j := n_{tasks}/3$;
For i := 1 to j **do**
Begin
$k := RandomInt(0, n_{tasks}) + 1$;
$L_t(T_k) := RandomInt(MinLevel, MaxLevel)$; /* $MinLevel$ and $MaxLevel$ are minimum and maximum values in the set L_t */
End;
Return(L_t); /* return the set of temporary levels */
End;

We now briefly go through the entire algorithm. The algorithm begins by getting a set of initial levels from procedure **GetInitialLevels**. This procedure takes two parameters. The G_r is the reverse graph and $S(P, R)$ the multiprocessor. It then does the scheduling using the local $est(T, p)$ as the decision function. All the tasks which have no predecessors are included in RTR set. For each task in RTR we find $est(T, p)$ by exhaustive search on all processors. We then start the task having the minimum value of $est(T, p)$.

This task is referenced by the name $T_{assigned}$. The **Schedule** statement starts this task on the processor on which its $est(T, p)$ was found. The $ct(T_{assigned})$ is calculated by adding $\mu(T_{assigned})$ to the task's est value. The RTR set is then tried to be updated and if any successor of $T_{assigned}$ is found which has all its predecessors scheduled then we include it in RTR. At the same time we remove $T_{assigned}$ from RTR. This process continues until all the tasks have been scheduled and RTR is again empty. We then return the completion time values CT as initial task levels

to the main algorithm.

The next procedure is **MakeSchedule**. It takes three parameters. They are L, the set of levels, G, the forward graph and $S(P,R)$ the multiprocessor. Everything here is just the same except that the decision function changes to $d(T) = l(T) - 100 \times est(T,p)$. At each scheduling step we assign the task which maximizes $d(T)$ from amongst the tasks in RTR. After scheduling all the tasks we return the complete schedule as solution to the main algorithm. The solution comprises of the start time, finish time and processor for each task.

The returned schedule or the seed schedule is called S_0. The schedules S and S_{best} are also initialized to S_0. The initial control parameter value is set to 1% of the finish time of S_0. The control parameter cp is set to cp_0. This in effect means that we allow schedules which are within 1% of the seed schedule to be accepted.

The function **Cost** takes the schedule S and returns the finishing time of the task which finishes last. The procedure **FindMobility** does the job of analyzing the schedule and finding mobility of each task in G. It takes three parameters. They are S, the generated solution or schedule, G, the forward graph and $S(P,R)$ the multiprocessor. It then pulls down each task towards the bottom of the schedule or towards the exit node while honoring the requirements of the communication edges. The processor assignment of each task remain same and for each we find $st_{pulled}(T)$. This is necessarily greater than or equal to $st(T)$. Mobility is then simply found by $M(T_i) := st_{pulled}(T_i) - st(T_i)$ where $i = 1 \ldots n_{tasks}$. The set M of mobilities is then

returned to the main algorithm.

The procedure **Suggest** takes two parameters. They are L, the set of levels which generated the present schedule and M, the set of mobilities of tasks in the present schedule. It then generates the set of temporary levels by $l_t(T_i) := l(T_i) - M(T_i)$ where $i = 1 \ldots n_{tasks}$. Lastly, it returns the set of temporary levels L_t to main.

The procedure **DoMutation** takes the set of temporary levels L_t as its parameter. It then randomly selects 3% of the tasks for mutations. The function $RandomInt(low, high)$ generates a random integer between low and $high$ with a uniform probability distribution. For the tasks selected for mutations, we assign a random value between $MinLevel$ and $MaxLevel$ as its new level. $MinLevel$ and $MaxLevel$ are the minimum and maximum level values in the set L_t. The temporary levels are then returned after mutations are done.

In the main algorithm we use the set L_t for scheduling the graph G on $S(P, R)$. This results in a temporary solution called S_t. The gain of the new solution is computed and if it is positive we store the new solution in S_{best}. Positive gain results if the new schedule length is lesser than the minimum schedule seen so far. Thus if the new solution is better we always accept it.

If the new solution is not better than the minimum seen so far we accept it probabilistically depending on how far it is from minimum and the value of cp. For this we generate a random real number between $-cp$ and 0. If the gain is larger then this value we accept the new set L_t as our working point. Here we

are probabilistically accepting bad moves also if they are within the limit cp. If somehow we get stuck in a local minima and $Cost(S_t) = C_{pre}$, we increase the control parameter by the function f which increases the cp by 20%. As soon as we get out of this minima we reset the cp to cp_0. When $MaxIter$ iterations are done we return the best solution obtained so far.

6.4 Performance Evaluation of Mobility Based Evolution Scheduling

The workload was the same as used for the results in the previous chapter. The results were compared with two other techniques as done earlier. The other two heuristics are CD/ETF and CD/ERDETF. The finish time which is the minimum of the three is found and is called ω_{best}. Percentage deviation from ω_{best} are then shown in the graphs. The results are shown for both the HC and FC topology. We also show the average least number of iterations required by the evolution method to get its minimum.

A typical graph is chosen to demonstrate how the best finish time varies with iterations. This is shown in Figure 6.2. In the early phase we see that there is rapid improvement in finish time. The improvement later tapers off and the last improvement is found at iteration number 53.

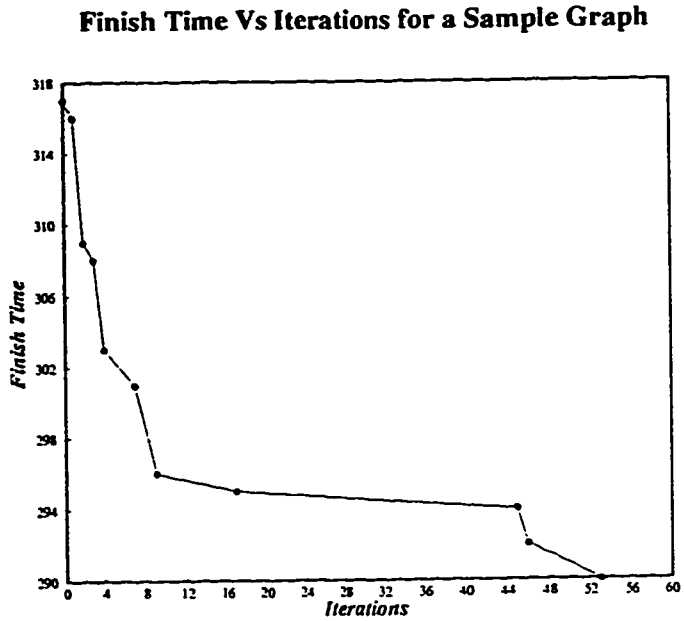


Figure 6.2: Finish time versus iterations for a typical graph.

6.5 Analysis of CD/ETF Results

The CD/ETF algorithm relies on the principle of load balancing. It tries to avoid idle time slots in a greedy way. We can say that it lacks a look ahead strategy. In the case of graphs having high parallelism this strategy works better. We can see from the results that with increasing parallelism the performance of CD/ETF becomes better.

When the parallelism is high there are many tasks competeing to get scheduled. Under these conditions the local maximising of processor efficiency gives good results. From Figure 6.3 we can see that the performance tends to degrade with increasing communication. This happens because if the communication is higher there is lesser chance of opverlapping computation with it. any wrong choices made

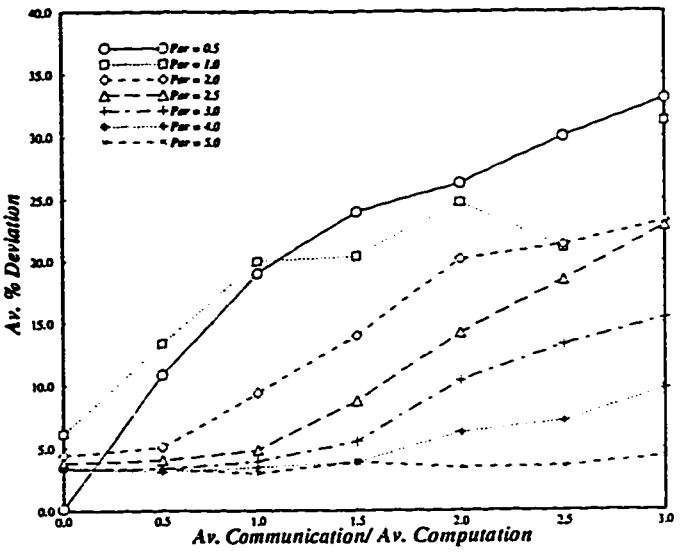


Figure 6.3: Percentage Deviation from best known solution of CD/ETF on FC topology.

by the heuristic will lead to a greater penalty because of increased communication. The performance at low parallelism and high communication can be as much as 32% away from ω_{best}. However at high parallelism values the deviation is around 4% from ω_{best}.

In the case of hypercube topology the effects are similar. The hypercube interconnection network just tends to increase the communication. So the figures are higher for it. The worst performance of CD/ETF is at low parallelism and high communication and it is 35% away from ω_{best}. At high parallelism degree the performance ranges from 4% to 11%. This is shown in Figure 6.4.

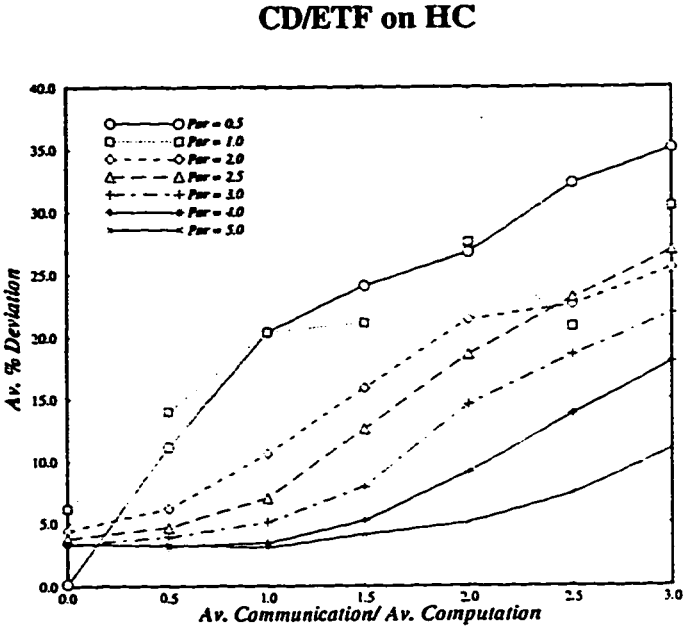


Figure 6.4: Percentage Deviation from best known solution of CD/ETF on HC topology.

6.6 Analysis of CD/ERDETF Results

The CD/ERDETF heuristic takes a very refined estimate of the criticality of each task. This ensures that its performance be much better than simple CD/ETF. For the fully connected topology the best performance is obtained at high parallelism degrees. From Figure 6.5 we can see that the deviation from ω_{best} in this case is 0 which means that it finds the minimum. At low parallelism degrees the deviation increases and the worst case is obtained when β is 2 and α is 3. This deviation is about 3.2%.

When the topology changes to hypercube the communication increases implicitly. This results in similar behavior but slightly degraded figures. The best performance

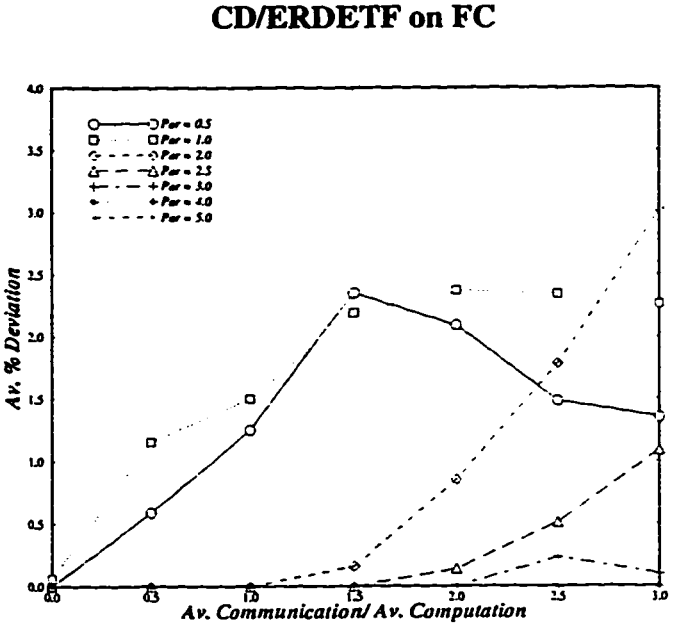


Figure 6.5: Percentage Deviation from best known solution of CD/ERDETF on FC topology.

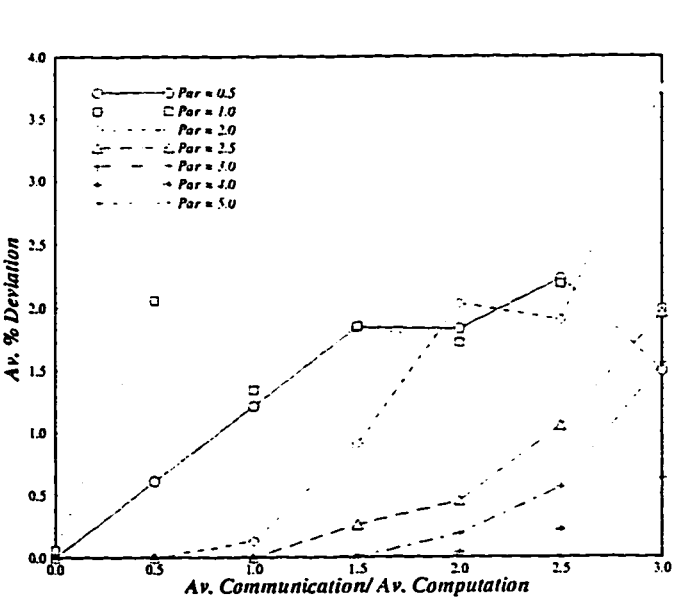


Figure 6.6: Percentage Deviation from best known solution of CD/ERDETF on HC topology.

is still at high parallelism and the deviation in this case is 0. The worst is again at $\beta = 2$ and $\alpha = 3$ and is about 3.7%. This is shown in Figure 6.6.

6.7 Analysis of Mobilty Based Evolution Scheduling ($MBES$) Results

The simulated evolution results are best amongst the three methods. In the case of low parallelism $\beta = 0.5$ and $\beta = 1.0$ the mobility based decision function is able to correctly discover critical tasks. The performance is good and deviation from ω_{best} is under 0.8% for the fully-connected topology. If the parallelism is high then also the performance is good. For the range $\beta = 4$ and $\beta = 5$ the deviation from ω_{best} is up to 1.1%. For medium parallelism the performance is not that good and varies up to 1.5%. This is shown in Figure 6.7. The performance at all values of parallelism degrades with increase in communication.

One important point to note is that the ETF effect is absent in this graph. The ETF effect gives better and better performance as the parallelism increases. This is because at high parallelism the crucial point is to keep the processors busy and this is what ETF does. In Figure 6.7 we see that the performance is not increasing with parallelism as say in Figure 6.5 or Figure 6.3. This is because of the mobility based decision function. At low parallelism it is essential to differentiate between critical tasks because the idle slot avoiding strategy of ETF will not work in this case. The

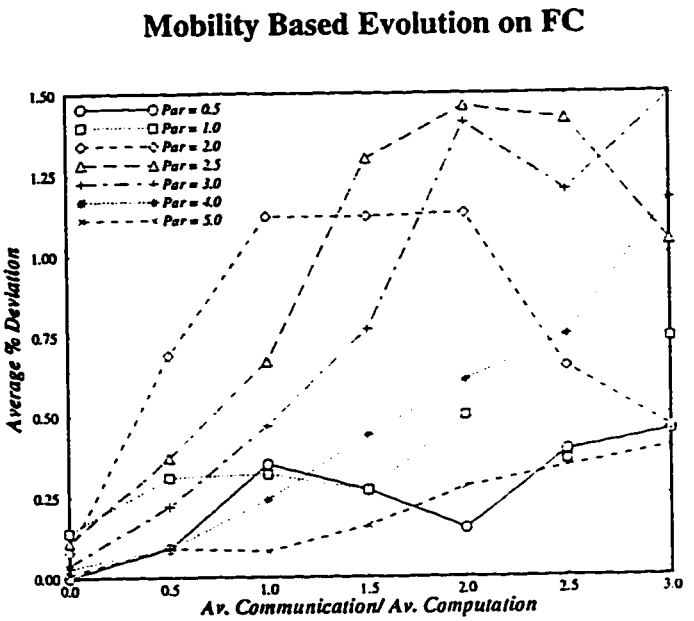


Figure 6.7: Percentage Deviation from best known solution of Mobility Based Evolution scheduling on FC topology.

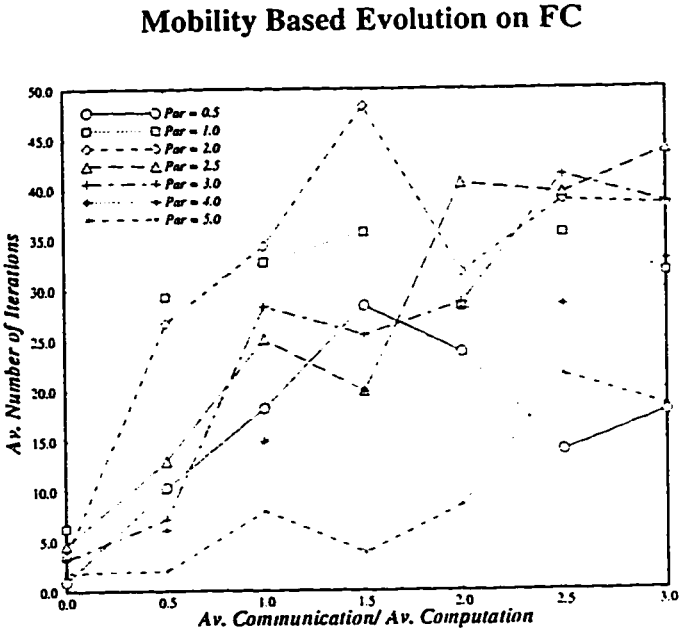


Figure 6.8: Average number of iterations required by Mobility Based Evolution scheduling on FC topology.

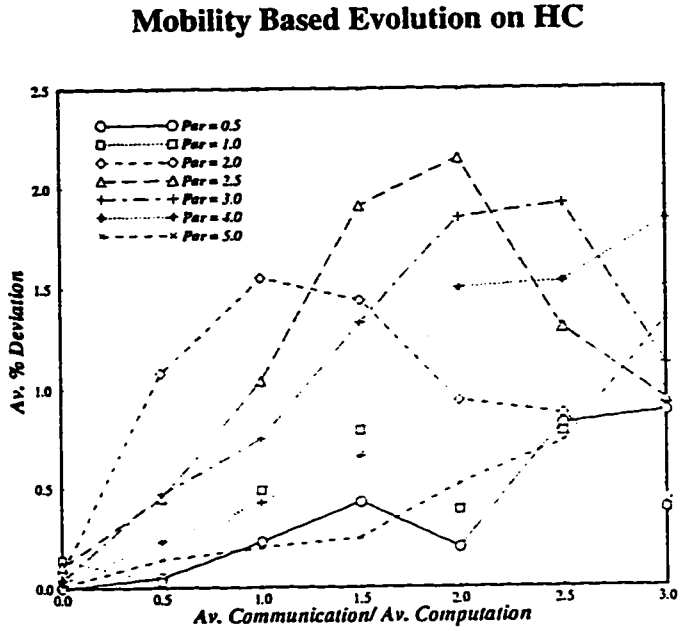


Figure 6.9: Percentage Deviation from best known solution of Mobility Based Evolution scheduling on HC topology.

mobility based decision function tackles this issue nicely. The best performance of MBES over CD/ETF is of about 34% for $\alpha = 3$ and $\beta = 0.5$.

The number of iterations needed by evolution are difficult to interpret. Generally with increase in communication the iterations required increase. Average number of iterations required at $\beta = 5$ are under 20. Maximum number of iterations are needed when parallelism and communication both are moderate. This figure is 48 for $\beta = 2$ and $\alpha = 1.5$. This is shown in Figure 6.8.

In the case of hypercube topology we see the same effects as for the fully-connected topology discussed above. The hypercube interconnection network only increases the network latency values and its effect is the same as increase in communication. We can see from Figure 6.9 that the trends are the same as for the

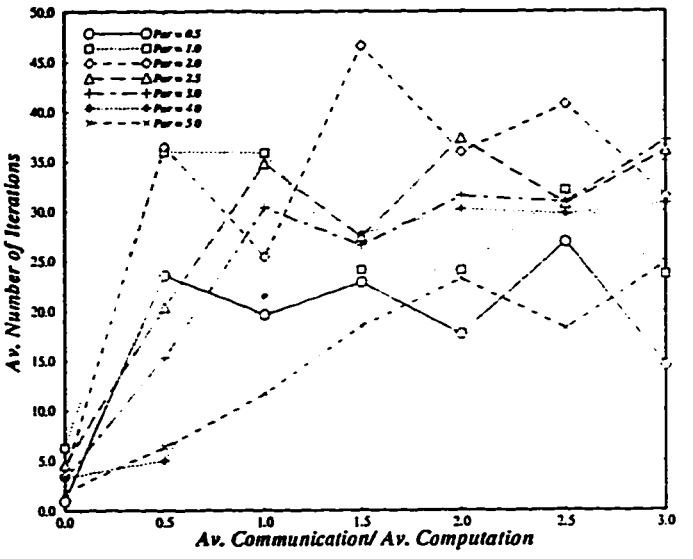


Figure 6.10: Average number of iterations required by Mobility Based Evolution scheduling on HC topology.

fully-connected case. The only difference is a slight vertical shift or a deterioration in performance. This performance as a whole is better than the other methods though. The worst performance here is again obtained at moderate parallelism and moderate communication and is 2.2% at $\beta = 2.5$ and $\alpha = 2$.

The average number of iterations needed to obtain the best result is shown in Figure 6.10. There is an increase in the amount of effort required with increase in communication at all parallelism values. All the results though can be obtained in under 48 iterations. This is very much desirable if the technique has to be applied in real environments like a parallelising complier.

The Tables 6.7 and 6.7 summarise the overall results across all values of parallelism and communications for fully-connected and hypercube topologies respec-

Heuristic	*% of Cases*	*% Deviation*
CD/ETF = CD/ERDETF	1.70	0.00
CD/ETF better than CD/ERDETF	0.06	0.23
CD/ERDETF better than CD/ETF	98.24	11.18
CD/ETF = MBES	1.70	0.00
CD/ETF better than MBES	0.00	0.00
MBES better than CD/ETF	98.30	11.26
CD/ERDETF = MBES	28.71	0.00
CD/ERDETF better than MBES	48.50	1.10
MBES better than CD/ERDETF	22.79	2.59

Table 6.1: Pairwise comparison between heuristics on fully-connected topology

Heuristic	*% of Cases*	*% Deviation*
CD/ETF = CD/ERDETF	1.70	0.00
CD/ETF better than CD/ERDETF	0.00	0.00
CD/ERDETF better than CD/ETF	98.30	12.88
CD/ETF = MBES	1.70	0.00
CD/ETF better than MBES	0.00	0.00
MBES better than CD/ETF	98.30	12.89
CD/ERDETF = MBES	21.90	0.00
CD/ERDETF better than MBES	51.77	1.45
MBES better than CD/ERDETF	26.33	2.79

Table 6.2: Pairwise comparison between heuristics on hypercube topology

tively. We again used 1470 graphs for each topology as for the testing of Arc Addition method. Percentage of cases for each of the three conditions during pairwise comparisons are given. We also give the average percentage deviation in the finish times at each comparison.

Chapter 7

Conclusions and Future Work

This chapter concludes this thesis and gives a brief summary of our two attempts at the scheduling problem. Suggestions regarding improvements and future work are also presented.

7.1 Conclusion

This research tackled the problem of scheduling precedence constrained computations with communication costs on distributed memory multiprocessor systems. The task graph or the computation problem is represented as a directed acyclic graph or *DAG*. This graph consists $(\Gamma, \rightarrow, \mu, C)$. The set Γ is a set of all the tasks of the task graph i.e. $(T_1, T_2, \ldots, T_m)$. The total number of tasks is m. The precedence constraints are represented by $\rightarrow$. Each task needs a specific amount of CPU time,

called the computation time and is represented by μ. Each precedence arc, $T_1 \rightarrow T_2$, carries a weight which denotes the amount of data communicated from T_1 to T_2. This is denoted by communication $c(T_1, T_2)$ and its set for the whole graph is C.

The multiprocessor is represented by $S(P, R)$. R represents the routing network and the routing costs. P represents the processors. A contention free media is assumed for the communication network of $S(P, R)$. The scheduling problem is defined as a mapping of the set Γ to the set of available processors so that the finish time is minimized. This problem is NP-Hard in its complexity.

A simulated evolution based approach to this problem was developed. The new approach uses task mobility as the definition of a task's criticality in a given schedule. The new method begins with a solution which is as much as 35% away from ω_{best}. The evolution based method then refines and modifies this solution to within 2.2% of ω_{best}. In many of the cases it outperforms an excellent heuristic, namely CD/ERDETF. The worst deviation of CD/ERDETF is about 3.8% from ω_{best}. For the new method this figure is 2.2% of ω_{best}.

The improved results are because of the improvement in the decision function. The mobility based decision function identifies the critical tasks in each schedule and tries to start them in a better position in the next schedule. The whole approach is implemented within the framework of simulated evolution algorithm. The number of iterations needed by the new technique are also not very large. In the worst case we may need up to 50 iterations on an average. This makes it attractive for use

in real parallelising compilers. The simulated evolution technique is an excellent mid way approach between pure search based methods and pure heuristic based methods. The results also prove the same.

7.2 The Arc Addition

The technique of arc addition is a powerful one. It is based on the idea of adding pseudo arcs. These arcs are then used to enforce two tasks connected by such an arc to be always together and follow a pre-defined precedence order as imposed by the new arc. The pseudo arc acts as a strong force between the two connected tasks. This enforcing of a particular order improves the results by as much as 5% with respect to ω_{best} when compared to CD/ERDETF. The excess effort required is in trying the pseudo arcs in the iterative process. This increases the time required to find the minimum schedule.

The draw back with the arc addition process was that we could not find increased performance by adding more than one arc. Because of this reason the simulated evolution part for this technique couldn't yield results.

7.3 Future Work

In the case of mobility based evolution scheduling we based our definition of task criticality on mobility of the task in the present schedule. There can be other

estimates of a tasks criticality and they should be investigated and tried. The decision function in our case uses only the level information and est information.

The est factor is the local factor and is given a weight denoted by factor κ. The value of this factor was chosen based on empirical results. Ideally there should be a theoretical basis to set its correct value. This aspect needs to be investigated further.

The performance curves for CD/EST heuristic show an interesting pattern. As the parallelism increases the performance of ETF becomes better. This is true to a lesser degree for other heuristics also. We argue that this happens because at high parallelism degrees the important point is to keep the processors busy and this is what EST achieves and hence the improved performance. This effect needs to be modelled mathematically so we can find the performance bounds for each decision function.

The multiprocessor in our case was assumed to be contention free. In reality this is seldom the case. We should therefore try to incorporate a probabilistic delay model for the latency of the network. This will make the results much more realistic.

The addition of a single pseudo arc was found to be enough to give an improved solution. The process of adding more than one arc to get an even more improved performance is attractive. The reason for not gaining any further improvement can be the lack of flexibility in the behavior of the pseudo arc. We should investigate other types of pseudo arcs as well.

The present version of pseudo arc imposes two conditions. If there is a pseudo arc, $T_1 \rightarrow T_2$, then T_2 will be started after T_1 and both will be scheduled on the same processor. This is maybe too restrictive. Other versions can be such that T_2 starts after T_1 finishes but it may be allowed to start on any processor. Also we can have that T_2 can start as soon as T_1 starts and on any processor. We can also have a combination of the different types of arcs for the same graph. With this additional flexibility we may get better results using simulated evolution.

Bibliography

[1] M.R. Garey, R.L. Graham, and D.S.. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, No 26:3–21, 1978.

[2] Kai Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, 1984.

[3] Israel Koren, Bilha Mendelson, and G. M. Silberman. A data-driven vlsi array for arbitrary algorithms. *IEEE Computer*, pages 30–43, October 1988.

[4] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Computing*, pages 244–257, Apr 1989.

[5] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Mathematics*, 17:416–429, 1969.

[6] E.G. Coffman et al. *Computer and Job-Shop Scheduling Theory.* John Willey and Sons, 1976.

[7] T. C.. Hu. Parallel sequencing and assembly line problems. *Operations Research*, No 9:841–848, May 1961.

[8] R.L. Graham and E. Coffman. Optimal scheduling for two-processor systems. *Acta Inform.*, 1:200–213, 1972.

[9] R. Sethi. Scheduling graphs on two processors. *SIAM Computing*, 5, No 1:73–82, Mar 1989.

[10] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[11] S. M. Sait and H. Youssef. *VLSI Physical Design Automation.* McGraw-Hill Book Company, 1995.

[12] Muhammad S.T. Benten and Sadiq M. Sait. Genetic scheduling of task graphs. *Int. J. Electronics*, 77, No 4:401–415, 1994.

[13] Edwin S. H. Hou, Nirwan Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems.* No 5:113–120, February 1994.

[14] S. Kirkpatrick, Jr. C. Gelatt, and M Vecchi. Optimization by simulated annealing. *Science*, No 220(4598):498–516, May 1983.

[15] Ralph Michael Kling and Prithviraj Banerjee. ESP: Placement by simulated evolution. *IEEE Trans. on Computer-Aided Design*, 8, No 3:245–256, Mar 1989.

[16] Ralph Michael Kling and Prithviraj Banerjee. Empirical and theoretical studies of the simulated evolution method applied to standard cell placement. *IEEE Trans. on Computer-Aided Design*, 10, No 10:1303–1315, Oct 1991.

[17] Youssef G. Saab and Vasant B. Rao. Combinatorial optimization by stochastic evolution. *IEEE Trans. on Computer-Aided Design*, 10, No 4:525–535, Apr 1991.

[18] Tai A. Ly and Jack T. Mowchenko. Applying simulated evolution to high level synthesis. *IEEE Trans. on Computer-Aided Design of Circuits and Systems*, 12, No 3:389–408, Mar 1993.

[19] T.L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *Comm. of the ACM*, 17, No 12:685–690, Dec 1974.

[20] M. Al-Mouhamed and A. Al-Maasarani. Performance evaluation of scheduling precedence-constrained computation on message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 5, No 12:1317–1322, December 1994.

[21] M. Al-Mouhamed and A. Al-Maasarani. Scheduling optimization through iterative refinement. In *PACT'95*, pages 178–184, Cyprus, 1995.

[22] H. M. Rashad Najjari. Investigation of optimization techniques for scheduling precedence computations with communication costs. Master's thesis, King Fahd University of Petroleum and Minerals, June 1996.

[23] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. *Proc. of the SIGPLAN Symp. on Compiler Construction*, pages 17–26, Jul 1986.

[24] T. Yang and A. Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5, No 3:951–967, Sep 1994.

[25] M.-Y. Wu and D.D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1, No 3:330–343, Jul 1990.

[26] Yu-Kwong Kwok and Ishfaq Ahmad. Scheduling task graphs on multiprocessor: Issues and a better algorithm. *To appear in IEEE Trans. on Parallel and Distributed Systems*, 1995.

[27] G.C. Sih and E.A. Lee. Scheduling to account for interprocessor communication within interconnection constrained processing network. *Inter. Conf. on parallel Processing*, I:9–16, 1990.

[28] B. Kruatrachue. Static task scheduling and grain packing in parallel processing systems. *Ph.D. Thesis, Department of Computer Science*, 1987. Oregon State University.

[29] Yu-Kwong Kwok and Ishfaq Ahmad. On using task duplication in parallel program scheduling. *Under review with IEEE Trans. on Parallel and Distributed Systems*, 1996.

Vita

Mohammad Mohsin Nadeem was born in Aligarh, India, on April 14, 1972. He completed his school education from Our Lady of Fatima, Aligarh. He received the degree of Bachelor of Science in Engineering with Electronics and Communication as major, in June 1993. He worked as a Research Assistant from January 1994 to December 1996 in the Computer Engineering Department of King Fahd University of Petroleum and Minerals. At the same time he completed his masters in Computer Engineering. He received the degree of Master of Science in Computer Engineering from the same university in December 1996.